

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

**A Thesis Submitted for the Degree of PhD at the University of Warwick**

<http://go.warwick.ac.uk/wrap/3059>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



**Creative Software Development:  
An Empirical Modelling Framework**

by

**Paul Edward Ness**

**Thesis**

Submitted to the The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of **Doctor of Philosophy**

**Department of Computer Science  
University of Warwick**

October 1997

# Contents

<b>List of Examples</b>	<b>viii</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>Acknowledgments</b>	<b>xiii</b>
<b>Declarations</b>	<b>xiv</b>
<b>Abstract</b>	<b>xv</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Preliminaries . . . . .	1
1.2 Aims and motivations . . . . .	2
1.3 Methodological challenges . . . . .	4
1.4 Sources of illustrative material . . . . .	5
1.5 Thesis outline . . . . .	7
1.6 Abbreviations . . . . .	9
<b>Chapter 2 Background to EM, PD and SD</b>	<b>11</b>
2.1 Empirical Modelling . . . . .	11
2.1.1 General concepts and principles . . . . .	12
2.1.2 Agentless systems . . . . .	17
2.1.3 Single-agent systems and modelling . . . . .	19
2.1.4 Computer as artefact . . . . .	21

2.1.5	Multi-agent systems and modelling . . . . .	23
2.1.6	Agent concept . . . . .	25
2.1.7	Conceptualization . . . . .	26
2.1.8	Situating EM . . . . .	28
2.2	Product design . . . . .	29
2.3	Software development . . . . .	32
<b>Chapter 3</b>	<b>Characterization of EM, PD and SD</b>	<b>38</b>
3.1	Background . . . . .	38
3.2	Artefacts . . . . .	39
3.3	Subjects . . . . .	45
3.4	Actions . . . . .	47
3.5	Constraints . . . . .	53
3.6	Environments . . . . .	56
3.7	Knowledge . . . . .	59
3.8	Summary and conclusion . . . . .	62
<b>Chapter 4</b>	<b>Artefacts of EM, PD and SD</b>	<b>65</b>
4.1	Definition . . . . .	65
4.2	Need for artefacts and how they help cognition . . . . .	67
4.3	Characterization of artefacts . . . . .	68
4.3.1	Novelty and familiarity . . . . .	71
4.3.2	Ambiguity and unambiguity . . . . .	72
4.3.3	Implicit meaningfulness and explicit meaning . . . . .	73
4.3.4	Emergence and completeness . . . . .	74
4.3.5	Incongruity and congruity . . . . .	75
4.3.6	Divergence and convergence . . . . .	76
4.4	Further characterizations of artefacts . . . . .	77
4.4.1	Construals . . . . .	77
4.4.2	Design drawings . . . . .	79
4.5	Summary . . . . .	82
	Appendix: Illustrative examples for Section 4.3 . . . . .	83



<b>Chapter 5</b>	<b>Actions of EM, PD and SD</b>	<b>97</b>
5.1	Definition . . . . .	97
5.2	Generative actions . . . . .	99
5.2.1	Retrieval . . . . .	100
5.2.2	Association . . . . .	101
5.2.3	Synthesis . . . . .	102
5.2.4	Transformation . . . . .	103
5.2.5	Analogical transfer . . . . .	103
5.2.6	Categorical reduction . . . . .	104
5.3	Exploratory actions . . . . .	105
5.3.1	Creative exploration and SD . . . . .	106
5.3.2	Attribute finding . . . . .	107
5.3.3	Conceptual interpretation . . . . .	108
5.3.4	Functional inference . . . . .	108
5.3.5	Contextual shifting . . . . .	109
5.3.6	Hypothesis testing . . . . .	110
5.3.7	Searching for limitations . . . . .	111
5.4	Further characterizations of actions . . . . .	112
5.4.1	Observation and experimentation . . . . .	112
5.4.2	Methods and methodology . . . . .	114
5.5	Summary . . . . .	115
	Appendix: Illustrative examples for Sections 5.2 and 5.3 . . . . .	117
<b>Chapter 6</b>	<b>SD as Systems Development</b>	<b>137</b>
6.1	Generalizing computers, programs and programming . . . . .	137
6.1.1	Computer as artefact . . . . .	138
6.1.2	Program as configuration . . . . .	141
6.1.3	Programming as configuring . . . . .	145
6.2	Addressing the essential difficulties of software . . . . .	149
6.2.1	Complexity . . . . .	149
6.2.2	Conformity . . . . .	151
6.2.3	Changeability . . . . .	152

6.2.4	Invisibility . . . . .	154
6.3	Attacks on the essential difficulties of software . . . . .	155
6.3.1	Buy versus build . . . . .	155
6.3.2	Requirements refinement and rapid prototyping . . . . .	156
6.3.3	Incremental development - grow don't build software . . . . .	158
6.3.4	Great designers . . . . .	159
6.4	Software and SD in the future . . . . .	160
6.4.1	Networked computing and concurrency . . . . .	161
6.4.2	Software agents . . . . .	162
6.4.3	Object standards and technology . . . . .	162
6.4.4	Product-oriented development . . . . .	164
6.5	Requirements engineering in the future . . . . .	165
6.5.1	Emerging importance of context . . . . .	165
6.5.2	End of requirements as contract . . . . .	168
6.5.3	Supporting market-driven inventors . . . . .	169
6.5.4	Coping with incompleteness . . . . .	170
6.5.5	Integrating design artefacts . . . . .	171
6.5.6	Making requirements methods and tools more accessible . . .	171
6.6	Conclusion . . . . .	172
6.7	Limitations of EM for developing software . . . . .	175
6.7.1	Quality . . . . .	176
6.7.2	Management . . . . .	177
6.7.3	Methodology . . . . .	179
6.7.4	Scale . . . . .	180
<b>Chapter 7</b>	<b>Conclusions and further work</b>	<b>182</b>
7.1	Conclusions . . . . .	182
7.2	Further work . . . . .	184
7.2.1	Software engineering . . . . .	184
7.2.2	EM in context . . . . .	188
7.2.3	Distribution of EM tools . . . . .	189

<b>Bibliography</b>	<b>191</b>
<b>Appendix A Empirical Modelling of a Sailboat</b>	<b>206</b>
A.1 Common-sense knowledge . . . . .	206
A.2 Agent-oriented modelling . . . . .	207
A.3 Observation-oriented modelling . . . . .	207
A.4 Definitive representation of the sailboat . . . . .	209
A.5 Exploring the sailboat simulation . . . . .	213
A.6 Extending the sailboat model . . . . .	214
<b>Appendix B Experiences Using the Shlaer-Mellor Method</b>	<b>217</b>
B.1 Overview of the Shlaer-Mellor method . . . . .	218
B.2 Domain analysis . . . . .	219
B.3 Using the Teamwork tool in analysis and design . . . . .	219
B.4 Automatic code generation . . . . .	220
B.5 Testing . . . . .	221
B.6 Conclusion . . . . .	222
<b>Appendix C SUL, MUL and Hydrolift Artefacts</b>	<b>223</b>
C.1 SUL artefacts . . . . .	223
C.1.1 SUL LSD specification . . . . .	223
C.1.2 SUL visualization/animation . . . . .	224
C.1.3 SUL DoNaLD script . . . . .	224
C.1.4 SUL ADM script . . . . .	227
C.1.5 SUL sketch . . . . .	228
C.1.6 SUL statement of requirements . . . . .	228
C.2 MUL artefacts . . . . .	229
C.2.1 MUL LSD specification . . . . .	229
C.2.2 MUL visualization/animation . . . . .	230
C.2.3 MUL DoNaLD redefinitions . . . . .	230
C.2.4 MUL ADM redefinitions . . . . .	230
C.2.5 MUL sketch . . . . .	231
C.2.6 MUL statement of requirements and models . . . . .	231



C.3	Hydrolift artefacts . . . . .	232
C.3.1	Hydrolift LSD specification . . . . .	232
C.3.2	Hydrolift visualization/animation . . . . .	233
C.3.3	Hydrolift DoNaLD redefinitions . . . . .	233
C.3.4	Hydrolift ADM redefinitions . . . . .	235
C.3.5	Hydrolift sketch . . . . .	236
C.3.6	Statement of requirements and models . . . . .	236
<b>Appendix D Reviews</b>		<b>238</b>
	Creative Cognition: Theory, Research, and Applications . . . . .	238
	Total Design: Integrated Methods for Successful Product Engineering	242
	Creating Innovative Products Using Total Design: The Living Legacy of Stuart Pugh . . . . .	244

# List of Examples

2.1	Observation and agent-oriented analysis in OXO . . . . .	14
2.2	Definitive representation of state in OXO . . . . .	16
2.3	0-agent view in OXO . . . . .	19
2.4	1-agent view in OXO . . . . .	20
2.5	Computer as artefact in OXO . . . . .	22
2.6	n-agent view in OXO . . . . .	24
2.7	Classification of agents in OXO . . . . .	26
2.8	Conceptualization in OXO . . . . .	27
2.9	Structure models in SD . . . . .	35
2.10	Behaviour models in SD . . . . .	35
2.11	Process models in SD . . . . .	36
3.1	LSD specifications in EM . . . . .	41
3.2	Visualizations in EM . . . . .	41
3.3	Animations in EM . . . . .	42
3.4	Sketches in PD . . . . .	43
3.5	Structure, behaviour and process models in SD . . . . .	44
3.6	Generative and exploratory actions in EM . . . . .	49
3.7	Generative and exploratory actions in PD . . . . .	50
3.8	Transformational actions in SD . . . . .	52
3.9	Statement of requirements as constraint in SD . . . . .	55
3.10	LSD specification as constraint in EM . . . . .	56
3.11	Traditional environments in design . . . . .	59

4.1	Familiarity in stating requirements . . . . .	83
4.2	Familiarity in EM . . . . .	84
4.3	Novelty in EM . . . . .	85
4.4	Ambiguity in stating requirements . . . . .	86
4.5	Unambiguity in SD . . . . .	87
4.6	Unambiguity in EM . . . . .	88
4.7	Ambiguity in EM . . . . .	89
4.8	Explicit meaning in SD . . . . .	90
4.9	Implicit meaningfulness in EM . . . . .	91
4.10	Emergence and completeness in EM . . . . .	92
4.11	Completeness in SD . . . . .	93
4.12	Incongruity and congruity in PD . . . . .	94
4.13	Incongruity and congruity in EM . . . . .	95
4.14	Divergence and convergence in EM . . . . .	96
5.1	Retrieval in EM . . . . .	117
5.2	Retrieval in PD . . . . .	118
5.3	Retrieval in SD . . . . .	119
5.4	Association in EM . . . . .	120
5.5	Association in PD . . . . .	121
5.6	Association in SD . . . . .	122
5.7	Synthesis in EM . . . . .	123
5.8	Synthesis in SD . . . . .	124
5.9	Transformation in SD . . . . .	125
5.10	Transformation in EM . . . . .	126
5.11	Transfer in EM . . . . .	127
5.12	Transfer in PD . . . . .	128
5.13	Transfer in SD . . . . .	128
5.14	Reduction in EM . . . . .	129
5.15	Reduction in SD . . . . .	130
5.16	Lack of incentive for exploration in SD . . . . .	131
5.17	Attribute finding in EM . . . . .	132



5.18	Conceptual interpretation in EM . . . . .	133
5.19	Interpretation in PD . . . . .	134
5.20	Functional inference in EM . . . . .	135
5.21	Contextual shifting in EM . . . . .	136
7.1	Organization of observations in EM . . . . .	187
7.2	Sequencing of observations in EM . . . . .	188
A.1	Representing the modeller in the SBS . . . . .	208
A.2	Representing the sail in the SBS . . . . .	210
A.3	Representing the rig and hull in the SBS . . . . .	211
A.4	Exploration in the SBS . . . . .	214
A.5	Emergence in the SBS . . . . .	215
A.6	Openness in the SBS . . . . .	216

# List of Tables

3.1	Elements of the product design specification . . . . .	54
4.1	Creative and analytical properties . . . . .	69
5.1	Generative and exploratory actions . . . . .	99
6.1	Contrasting complexity in software and systems. . . . .	150
6.2	Contrasting conformity in software and systems. . . . .	152
6.3	Contrasting changeability in software and systems. . . . .	153
6.4	Contrasting invisibility in software and systems. . . . .	154
6.5	EM themes associated with buy versus build. . . . .	155
6.6	EM themes associated with requirements and prototyping. . . . .	157
6.7	EM themes associated with incremental development. . . . .	158
6.8	EM themes associated with great designers. . . . .	160
6.9	Patterns of large software systems: failure and success. . . . .	176

# List of Figures

2.1	Tools and notations in EM . . . . .	17
2.2	tkeden being used to model OXO game . . . . .	22
2.3	Activity model for total design . . . . .	30
3.1	Baltimore and Ohio Railroad drafting room 1899 . . . . .	59
3.2	Comparison of EM, PD and SD . . . . .	63
5.1	Geneplore Model . . . . .	98
5.2	Lift system components . . . . .	134
D.1	Geneplore Model . . . . .	240
D.2	Activity model for total design . . . . .	243

# Acknowledgments

I would like to thank my friend and supervisor, Meurig Beynon, for his support and guidance throughout the research and writing of this thesis and for his infectious enthusiasm for my ideas.

I would also like to thank my friends, both inside and outside the department, who made the good times great and the bad times bearable over the last few years, Anna Philippou especially.

Special thanks goes to my mother, father and brother for the help they have given me in so many different ways. I am indebted to my brother Steve who took on the unpleasant task of introducing me to computing.

Finally, thanks goes to the EPSRC and IBM who funded the research through a CASE award and to Hugh Darwen and Steve Russ for making it happen.

\* \* \* \* \*

Further thanks goes to Mike Holcombe for his constructive comments during the viva and to Meurig Beynon and Steve Russ for guiding me through the revisions.

# Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

The various aspects concerning the Empirical Modelling of the sailboat have been published in [NBY94]. The view of Empirical modelling expressed in this thesis has been represented in [BNR95]. The view of Empirical Modelling as a new paradigm for computing has been presented in [BN95].



# Abstract

The commercial success of software development depends on innovation [Nar93a]. However, conventional approaches inhibit the development of innovative products that embody novel concepts. This thesis argues that this limitation of conventional software development is largely due to its use of analytical artefacts, and that other activities, notably Empirical Modelling and product design, avoid the same limitation by using creative artefacts. Analytical artefacts promote the methodical representation of familiar subjects whereas creative artefacts promote the exploratory representation of novel subjects. The subjects, constraints, environments and knowledge associated with a design activity are determined by the nature of its artefacts.

The importance of artefacts was discovered by examining the representation of different kinds of lift system in respect of Empirical Modelling, product design and software development. The artefacts were examined by identifying creative properties, as characterized in the theory of creative cognition [FWS92], together with their analytical counterparts. The processes of construction were examined by identifying generative and exploratory actions. It was found that, in software development, the artefacts were analytical and the processes transformational, whereas, in Empirical Modelling and product design, the artefacts were both creative and analytical, and the processes exploratory.

A creative approach to software development using both creative and analytical artefacts is proposed for the development of innovative products. This new approach would require a radical departure from the established ideas and principles of software development. The existing paradigm would be replaced by a framework based on Empirical Modelling. Empirical Modelling can be thought of as a situated approach to modelling that uses the computer in exploratory ways to construct artefacts. The likelihood of the new paradigm being adopted is assessed by considering how it addresses the topical issues in software development.



# Chapter 1

## Introduction

This chapter starts with the aims and motivations for the thesis and associated research. It includes an account of the methodological challenges the subject of the thesis presents for research, challenges that had to be met by the author and that the reader should be aware of to fully appreciate the work. The source of the illustrative material used in the thesis is indicated. The chapter finishes with an outline of the thesis and abbreviations.

### 1.1 Preliminaries

A reader unfamiliar with the approach to modelling, known as Empirical Modelling, is recommended to see Appendix A which gives an account of the author's first experiences of Empirical Modelling when he modelled a sailboat as part of the M.Sc. module *Definitive Methods for Concurrent Systems Modelling* in December 1992 at the Department of Computer Science, University of Warwick. The account should give anybody who is unfamiliar with the modelling approach useful insight into the main concepts, principles, heuristics, techniques, notations and tools which constitute Empirical Modelling. For a more objective analysis of the modelling of the sailboat the reader is invited to read the paper [NBY94] which is an elaboration of the personal account for publication.

## 1.2 Aims and motivations

The motivation for this thesis stems from the interest expressed by the now disbanded IBM Warwick Software Development Laboratory (WSDL) in the potential of Empirical Modelling as a software development method. At about the same time they expressed an interest in Empirical Modelling the laboratory was changing its approach to software development from a traditional one, based on workbooks, to the established Shlaer-Mellor object-oriented analysis and design method [DSWW93, SM88, SM92]. Appendix B is a report prepared after an interview, by the author in November 1993, with those who were considering the prospect of giving up the old approach and adopting the techniques of the new method. This report helps to clarify the context in which the ideas developed in this thesis originated.

The primary aim of this thesis is to investigate the suitability of Empirical Modelling as a framework for a new approach to software development [BRS<sup>+</sup>89] that has creative as well as analytical components.<sup>1</sup> It was found inappropriate, during the investigation of Empirical Modelling, to think of Empirical Modelling in terms of conventional software development. Empirical Modelling provides general concepts and principles which apply to creative as well as analytical activities and artefacts. Showing Empirical Modelling as an appropriate framework for software development has meant finding ways of being explicit about how it combines creative and analytical components and the role that creativity has in the development of software.

In addition to the primary aim, a number of important issues in Empirical Modelling and software development were identified for the research to address:

**Empirical Modelling as product design** To investigate how Empirical Modelling relates to an essentially creative discipline, such as conceptual design.

This investigation serves two purposes. First, to validate the assumption that Empirical Modelling is an appropriate vehicle for creativity. Second, to avoid

---

<sup>1</sup>The term *components* of Empirical Modelling and software development is used in a specific sense to mean the artefacts and the actions used by modellers and software developers in order to construct artefacts. The terms *creative* and *analytical* qualify the nature of the components: creative components are novel artefacts and the actions of creating novel artefacts; analytical components are familiar artefacts and the actions of forming familiar artefacts. Further clarification of these terms is left for subsequent chapters.



the framework for creativity becoming biased towards analysis; there is a danger that the general concepts and principles of Empirical Modelling may acquire analytical connotations if they are related to software development only.

**Integrity of Empirical Modelling** To investigate creativity in the Empirical Modelling framework in a way that has scientific integrity. Creativity has always been considered unresearchable because it lacks the rigour thought necessary for science [FWS92]. It tends to depend on introspective and descriptive accounts of creative processes. However, an approach is needed in order to relate Empirical Modelling, product design and software development that has a degree of scientific integrity appropriate to a thesis written for the scientific community. This methodological issue seldom arises when investigating analytical components of the traditional software development framework where the activities of software developers and the necessary properties of formal artefacts are well-established.

**A paradigm for creative software development** To propose a paradigm for creative software development and assess the likelihood of it being adopted by those currently involved in software development. This proposal is based on the Empirical Modelling framework and borrows from the concepts and principles of Empirical Modelling, product design and conventional software development. Suggestions for alternative paradigms of software development tend to be conservative. A paradigm based on a framework that includes creative and analytical components would be radically different from the paradigm of conventional software development and could potentially change the way software development is thought about and done in the future.

**Resolving the software crisis** To challenge the assumption that the software crisis will be resolved with the emergence of a software engineering discipline that takes an analytical approach to software development. Gibbs [Gib94] cites the view of many in software development who are confident they are on course for the end of the software crisis believing it is essentially a matter of time before software development and computer science merge to provide a foundation



for software engineering. This raises a question: will software development and computer science merge in the future and if they do merge will software engineering provide a total solution to the software crisis?

### 1.3 Methodological challenges

The subject of this thesis has been regarded as largely unresearchable by scientists and engineers [FWS92]. Although creativity in product design and software development has long been a topic of interest it has not been considered a serious subject for study. Finke *et al* [FWS92] identify two primary reasons for this attitude. One is that the subject of creativity has had unscientific connotations, perhaps resulting from the reliance on anecdotal and introspective accounts to describe the creative process. Textbooks, if they mention creativity at all, tend to do so in an informal, descriptive way compared with the rigour given to processes in science and engineering. The other reason is the difficulties of studying creativity under controlled conditions. It is generally accepted that the creative process depends on the situation in which it occurs and cannot therefore be isolated for the purposes of description and rationalization without losing some of its essential qualities.

Finke *et al* address both these problems by adopting an experimental approach to the study of creativity. They present their approach in a book entitled “Creative Cognition: Theory, Research, and Application” reviewed in Appendix D. Experiments address the problem of scientific integrity by lending objectivity to anecdotal and introspective accounts of creative processes. Experiments, when properly executed, involve the rigorous testing of hypotheses by a community of scientists [Kap64, And68, CM81]. In addition, experiments address the problem of situatedness by allowing creativity to be investigated *in situ* rather than bringing it into the domain of mathematics and logic that is typical of theoretical approaches.

The author of this thesis addresses the problems of scientific integrity and situatedness in a similar way to Finke *et al* [FWS92] by taking an experimental approach. Although such an approach is unusual in computer science it was found to be an appropriate way of researching creativity with respect to the development of software. This thesis can be viewed as the result of an experimental approach



to the research of creativity that began with hypotheses stated as aims earlier in this chapter. Research continued by observing Empirical Modelling, product design and software development in order to test the hypotheses. The results of observing the activities and the conclusions drawn from the results are given in subsequent chapters.

Taking an experimental approach to researching creativity causes problems when it comes to representing the results. There is not necessarily an established conceptual framework in which the scientist can work as there is in, for example, computer science. Finke *et al* addressed this problem by using accounts of experiments, drawings of structures, illustrative examples, case studies and descriptions of situations. In this thesis the author has made similar use, as for example in describing Empirical Modelling, of accounts of modelling, pictures of models, illustrative examples, case studies and descriptions of modelling situations in order to support the claims made in the main text. A similar approach to presentation is taken in other literature on Empirical Modelling [BNR<sup>+</sup>89, BBY92, BFY93, NBY94, BJ94, BSY95, BC97].

## 1.4 Sources of illustrative material

The main source of illustrative material for this thesis was an extensive project on lift systems. The lift project involved Empirical Modelling, product design and software development for conventional lift systems as well as an unconventional hydraulic system called a Hydrolift. It also involved the discovery of similar work going on at Stanford University under the name of the Sisyphus project [RGE<sup>+</sup>94, Yos92, Yos94] to do with research in ontologies for knowledge representation for engineering lift systems. Most of the work has been done by the author with some notable exceptions: Suker and Sidebotham [SS94] were funded by the University of Warwick to work during the summer on the lift project, and Yung and Joy worked on relating the Sisyphus models to the Empirical Modelling framework.

The lift project began with the author creating models using Empirical Modelling and object-oriented analysis to represent a conventional lift system from the viewpoint of an individual using the lift. The models based on this personal view-



point were given the name of single-user-lift or SUL models. A public-domain CASE tool called OOTher [Zie94], based on similar commercially available tools, such as Teamwork used at the IBM WSDL (Appendix B), was used to create the SUL object-oriented models. The object-oriented models were implemented according to conventional object-oriented design and coding approaches using a generic architecture supporting graphical user interfaces and translator, both developed by the author.

The next important stage of the lift project came when the author worked with Suker and Sidebotham on extending the empirical models. First, the group used Empirical Modelling to represent a conventional lift system from the combined viewpoints of many individuals using a lift. The models based on this multiple viewpoint were given the name of multi-user-lift (MUL) models. Second, the group used Empirical Modelling to represent an innovative concept for a lift system, called a Hydrolift, from the viewpoint of an engineering designer. The models based on this engineering viewpoint were given the name of Hydrolift models.

Working in a group meant that communication between members emerged as an important issue. The group members used their LSD specifications, visualizations and animations to communicate their ideas to one another. Occasionally, these conventional artefacts of Empirical Modelling were supplemented by sketches such as a product designer might create during conceptual design. These sketches are reproduced in this thesis to illustrate the artefacts of product design. Sketches were particularly useful in communicating ideas about the Hydrolift.

The object-oriented MUL and Hydrolift models were created by the author after the group project. The author created the structure, behavioural and process models by following a method that was a generalization of the standard object-oriented methods. This began by describing the MUL and Hydrolift as a statement of requirements and then transforming the statement into the appropriate models. Although the author did not use the CASE tool OOTher for this exercise he did use the general principles that the tool shares with other commercially available, and typically more powerful, tools.

Other sources of material used in this thesis include a number of significant



Empirical Modelling projects:

- a sailboat resulting in a sailboat simulator (SBS) [NBY94] (Appendix A);
- a vehicle cruise controller (VCCS) [BBY92];
- the interaction between pupils and teacher in a classroom [Dav96];
- construction of a suite of OXO-like games [BJ94] (Chapter 2);
- a railway system [ABCY94c];
- the construction of a jigsaw puzzle [BSY95];
- the behaviour of a digital watch [BC95].

These projects did not involve the author in any direct way, except the SBS that was done almost entirely by the author. It is always made clear when material from these projects is used in the thesis.

## 1.5 Thesis outline

This thesis draws extensively on the concepts and principles of Empirical Modelling, product design and software development. The principal aim of this thesis is to investigate the suitability of Empirical Modelling as a framework for software development. Product design provides a context to this investigation to ensure that both the creative and analytical aspects of Empirical modelling are considered.

Chapter 2 introduces Empirical Modelling, product design and software development. It presents the basic concepts and principles used throughout the thesis. Empirical Modelling is introduced as a situated computer-based approach to modelling developed at the University of Warwick by the Empirical Modelling Group; the account of product design is based on Pugh's concept of total design [Pug91], with emphasis on the conceptual design phase; a mainstream view of software development is adopted, focusing on object-oriented analysis.

An insight into how Empirical Modelling, product design and software development differ in character is achieved by comparing them. However, it is important

to find an appropriate framework for comparison because the three activities are very different in nature. Consideration of artefacts forms a suitable basis for such a framework. The various aspects of Empirical Modelling, product design and software development are then compared with respect to their use of artefacts.

Chapter 3 compares Empirical Modelling, product design and software development. The comparison focuses on the artefacts: the LSD specification, visualization and animation in Empirical Modelling, the sketch in product design, and the structure, behaviour and process models in software development. In addition to artefacts, the actions, subjects, constraints, environments and knowledge of the activities are compared: knowledge informs actions on an artefact within an environment to represent a subject under certain constraints.

Chapter 3 highlights the importance of artefacts in determining the nature of Empirical Modelling, product design and software development. Further investigation of the nature of these artefacts requires an understanding of what makes them essentially different. This investigation demands a framework for identifying and contrasting the properties of artefacts.

Chapter 4 compares the artefacts of Empirical Modelling, product design and software development to identify how they are essentially different. A framework is provided by a set of creative properties, characterized in the theory of creative cognition [FWS92], and their complementary analytical counterparts. The results of examining the artefacts of the lift project with respect to each of the properties are given. The characterization of artefacts is extended to construals for representing novel phenomena [Goo90] and engineering drawings [Fer92].

The activities of Empirical Modelling, product design and software development are essentially sequences of situated actions performed on artefacts by modellers, designers and software developers. Chapter 3 argued that the character of actions and other aspects of activities is determined by artefacts. It follows that the nature of the artefacts identified in Chapter 4 can be expected to determine the nature of actions.

Chapter 5 compares the actions of Empirical Modelling, product design and software development to identify how they are essentially different. A suitable frame-



work for comparison is provided by the theory of creative cognition [FWS92] that characterizes processes as generative and exploratory. The results of examining the processes of the lift project with respect to each kind of action are given. The characterization of processes is extended to observation and experimentation in scientific inquiry [Kap64].

Chapters 3, 4 and 5 show that the artefacts and actions of Empirical Modelling and product design are different from those of software development. This suggests that Empirical Modelling and product design cannot be used as an approach to developing software in the conventional sense. Empirical Modelling and product design are more appropriately applied to the creative development of innovative systems than the methodical transformation of requirements into software that characterizes software development. One way that Empirical Modelling and product design could be construed as approaches to developing software is if software development could be viewed as systems development.

Chapter 6 considers how Empirical Modelling and product design might be used as an approach to developing software based on a generalization of the notions of computer, program and programming. The conventional view of the computer as an electronic device, or embodiment of a Turing machine, is generalized to *computer as artefact*, program as stored program is generalized to *program as system configuration*, and programming as software development is generalized to *programming as configuring systems* that is essentially the activity of Empirical Modelling and product design. The usefulness of this alternative view is assessed by considering how it addresses the topical issues in software development and requirements engineering.

Chapter 7 draws conclusions from the discussions and results in Chapters 3,4,5 and 6 and addresses the aims given in Chapter 1 and finishes with suggestions for further research in the area of creative software development.

## 1.6 Abbreviations

Abbreviations have been used throughout the thesis to improve readability:

**ADM** Abstract Definitive Machine and associated definitive programming language;

**CDS** Component Design Specification;

**DoNaLD** Definitive Notation for Line Drawing.

**EDEN** General purpose definitive programming language (abbreviation of “Evaluator of DEfinitive Notations”);

**EM** Empirical Modelling;

**Hydrolift** The Hydrolift system from the viewpoint of a lift engineer;

**LSD** Specification language used in Empirical Modelling;

**MUL** Multi-user-lift system or, in other words, a conventional lift system from the combined viewpoints of multiple users;

**PD** Product design;

**PDS** Product Design Specification;

**SD** Software development;

**SUL** Single-user-lift system or, in other words, a conventional lift system from the viewpoint of a single user.



## Chapter 2

# Background to EM, PD and SD

This thesis draws extensively on the concepts and principles of Empirical Modelling, product design and software development. The principal aim of this thesis is to investigate the suitability of Empirical Modelling as a framework for software development. Product design provides a context to this investigation to ensure that both the creative and analytical aspects of Empirical modelling are considered.

This chapter introduces Empirical Modelling, product design and software development. It presents the basic concepts and principles used throughout the thesis. Empirical Modelling is introduced as a situated computer-based approach to modelling developed at the University of Warwick by the Empirical Modelling Group; the account of product design is based on Pugh's concept of total design [Pug91], with emphasis on the conceptual design phase; a mainstream view of software development is adopted, focusing on object-oriented analysis.

### 2.1 Empirical Modelling

Empirical Modelling (EM) is a new approach to computer-based modelling that has emerged in the last few years during research by the Empirical Modelling Group at the University of Warwick. The choice of epithet *empirical* reflects the fact that our approach is rooted in observation and experiment. The emphasis on modelling what is experienced rather than preconceived distinguishes our approach from traditional approaches to computer-based modelling.

This section draws on the experiences of modelling an OXO game to illustrate EM. For further details of modelling a game of OXO the reader is invited to read the paper [BJ94] which is an elaborated account for publication. The reader may also wish to refer to the account of modelling a sailboat in Appendix B whilst reading this section.

### 2.1.1 General concepts and principles

Although EM is quite a recent innovation it is the result of long-term research into a particular kind of modelling called *agent-oriented modelling over the definitive representation of state* (AOMDRS). The concepts and principles specific to EM will be introduced later in this section. Before doing so the main concepts and principles underlying AOMDRS will be described.

There are three key concepts in AOMDRS: observable, agent and dependency. These concepts are defined as follows:

- an **observable** is any feature of the subject or model that can be reliably perceived, identified, and compared with similar features [Rus97];
- an **agent** is anything (human or otherwise) capable of changing the state of the subject or model [Rus97];
- a **dependency** is a relation between observables such that changing the value of a certain observable has a predictable effect upon the values of other observables.

The key notions of observable, agent and dependency are interrelated, thus providing a cohesive conceptual framework to AOMDRS: an observable has meaning with reference to the perception and interaction with a feature of the subject or model by an agent; a state has meaning with reference to simultaneous observations made by an agent [BRY90]; the simultaneous observation of observables gives the concept of dependency and state its meaning; action by agents is mediated through dependencies between observables [Bey97].

The central principle of AOMDRS is to establish a correspondence between two sets of observables: the real-world observables that represent the subject and the



reference set of observables that is defined by the features of the model. Establishing the correspondence is an iterative process, leading to successive refinement of the model and circumscription of the subject, that involves observation and experiment within the model and situation. The objective in setting up this correspondence is to achieve consistency between the way in which sets of observables are indivisibly linked in change in the subject and the way in which the corresponding sets of observables are indivisibly linked in change in the model [BNR95].

The correspondence between subject and model is achieved in AOMDRS by the modeller using the complementary techniques of observation and agent-oriented analysis and definitive representation of state:

- **Observation and agent-oriented analysis** involves the identification of observables, agents and dependencies in the subject, as shown in Example 2.1. This is achieved through interaction by the modeller with the subject and model in parallel. Through interaction the modeller reconciles what they observe of the subject with their beliefs about the subject as represented in the computer model. The modeller's beliefs about the subject are also recorded as a document in an LSD specification [Bey86b].

An LSD specification details the observables whose values can act as stimuli for an agent (its oracles), that can be redefined by the agent in its responses (its handles), those observables whose existence is intrinsically associated with the agent (its states) and those indivisible relationships between observables that are characteristic of the interface between the agent and its environment (its derivatives). The repertoire of possible state-changing actions of agents is also recorded (its protocol) [Bey97].

An LSD specification generally admits many different operational interpretations, corresponding to different presumptions about the environment in which agents interact, and the nature and reliability of their stimulus-response patterns. The LSD specification can be given an operational meaning within the framework of the Abstract Definitive Machine (ADM) [Sla90]. In the ADM, transitions are represented by parallel redefinition of variables in a definitive script. In this context, the user can interact freely with the model as a supra-

gent both to introduce new redefinitions on-the-fly and to dictate the pattern of agent-interaction.

---

**Example 2.1. Observation and agent-oriented analysis in OXO.** Four agents can be identified by the modeller in the game of OXO:

- the board;
- the player who places Xs on the board;
- the player who places Os on the board;
- the umpire who decides who plays next and who has won.

The player agents are the easiest to identify because these are the roles played by the modeller in games of OXO. The player interacts with the board during play even though the status of the board as an agent is somewhat obscure (Example 2.7). The decision by the modeller to identify an umpire agent indicates that the modeller views playing and the rules of play as conceptually distinct.

The modeller defines each of the agents in LSD. The LSD definitions of the board, player and umpire agents could be as follows:

```
agent board() { state Board = BlankBoard }

agent player(P, 0) {
state      choice
oracle     turn    Board
handle     Board
protocol
  turn == P && available(Board, choice) -> take(Board, choice),
  !available(Board, choice) -> make_new_choice()
}

agent umpire() {
state      turn
oracle     numofX   numofO
handle     Board
derivate
  turn = (numofX > numofO) ? player_X : player_O
protocol
  win(player_X) -> congratulate(player_X); Board = BlankBoard
  win(player_O) -> congratulate(player_O); Board = BlankBoard
  tie() -> declare_tie(); Board = BlankBoard
}
```

These agent definitions record the observations and relationships between observations identified by the modeller whilst playing OXO. The definitions are personal (what is involved in making a new choice?) and subject to revision (the umpire needs to know who played first in order to determine whose turn it is when there are an equal number of noughts and crosses on the board).

---



- **Definitive representation of state** is the the process whereby the modeller represents observables by variables and introduces definitions (similar in character to the defining of formulae for cells of a spreadsheet) to express the way in which the values of observables in the subject are interdependent, as shown in Example 2.2. Such a set of definitions - a definitive script - represents a possible experimental observation, and redefining a variable corresponds to changing an experimental parameter [ABCY94c].

A language that may be used to write a definitive script is referred to as a definitive notation. Each definitive notation is conceived with a mode of visualization in mind. Each has its own set of data types and underlying algebra, appropriately chosen for the scope of the application. ARCA, DoNaLD and SCOUT are examples. ARCA [Bey86a] was designed for the display and manipulation of combinatorial diagrams, DoNaLD (Definitive Notation for Line Drawing) [ABH86] for two-dimensional line drawing and SCOUT [Dep92] for describing screen layouts. To complement these special purpose notations, the definitive language EDEN [YY88, Yun90] incorporates C-like data types and operators to facilitate more general applications and the implementation of other definitive notations [BYCH92].

Figure 2.1 shows the relationship between the tools and notations used in observation and agent-oriented analysis and the definitive representation of state. The `tkeden` interpreter, used to implement the ADM, is discussed in Section 2.1.4.

AOMDRS is based upon the concepts of observable, dependency and agent, and the principles of analyzing observables and agents and the definitive representation of state as described above. However, these descriptions are unlikely to satisfy those who are looking for a systematic approach to the analysis and precise representation of systems. Most would probably accept that the concepts and principles have an intuitive meaning and that they can be interpreted in many different ways. In particular, there is no formal definition of observable, dependency and agent. There is no systematic way of identifying these in the subject either.



---

**Example 2.2. Definitive representation of state in OXO.** In parallel with the definition of LSD agents the modeller can translate the partial agent definitions into ADM entities and can implement the entities as definitive scripts. The translation essentially replaces derivatives by definitions and protocols by actions. This process requires the modeller to address issues of synchronization between observables that emerge when the LSD specification is interpreted operationally.

The model of the OXO game is developed incrementally through a sequence of modelling steps, each of which leads to the construction of a definitive script, named as an EDEN file, capturing assumptions about the synchronization of observations as identified during agent and observation-oriented analysis (Example 2.1):

- What is the geometry of the board ? (`geometry.e`)
- How does the actual board and what I perceive conform? (`display.e`)
- Can I interpret the board in OXO terms? (`status.e`)
- What considerations guide me in contemplating the next move? (`sqvals.e`)
- Whose turn is it to play? (`gamestate.e`)

This hierarchical organization of the modelling reflects the hierarchy of perceptions and actions underlying OXO-playing, ranging from low-level capabilities to see the board and apprehend geometric patterns to high-level abilities to interpret positions and apply rules [BJ94].

The final stage of model construction involves automating one of the players in the game of OXO. The ADM player entity translates into an EDEN triggered action (`control.e`) that takes a turn at OXO.

Each of the files of definitions correspond to the modeller's answer to the questions posed by observations and agency identified in the LSD specification. For example, the file `geometry.e` contains the following EDEN definitions:

```
allsquares is [s1,s2,s3,s4,s5,s6,s7,s8,s9]
lin1 is [s1,s2,s3]
lin2 is [s4,s5,s6]
...
linesthru1 is [lin1,lin4,lin7]
linesthru2 is [lin1,lin5]
...
linesthru is [linesthru1, linesthru2, ..., linesthru9]
```

These definitions indicate the decision by the modeller to represent the board geometrically in terms of lines of squares. This corresponds to how the modeller perceives the board whilst playing the game of OXO.

---



The original aim of this thesis - to investigate the suitability of our approach to modelling as a SD method - made it necessary to find a more formal definition of AOMDRS. The search for a more formal definition and the issues that emerged during the search contributed to the emergence of EM. EM has AOMDRS as its basis with additional concepts and principles that give EM integrity and distinguish it from other approaches to computer-based modelling. The uniqueness of our approach to computer-based modelling results from the emphasis on modelling what is experienced rather than what is preconceived.

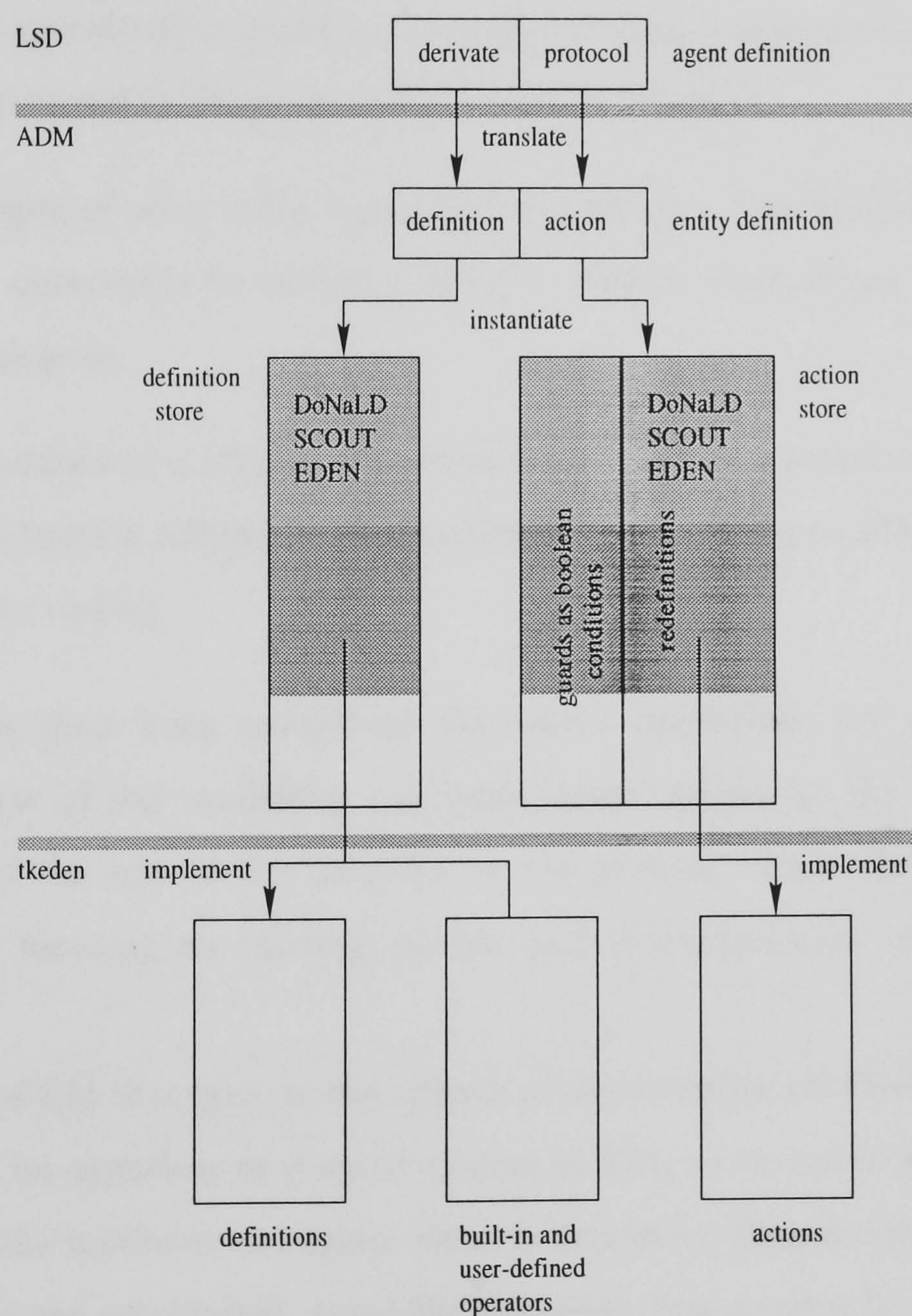


Figure 2.1: Tools and notations in EM.

### 2.1.2 Agentless systems

The original aim of this thesis was motivated by the apparent similarity between our approach to modelling and the Shlaer-Mellor method of object-oriented analysis



(SMOOA) and design (SMOOD) [SM88, SM92] as was being used at the IBM WSDL (Appendix B):

- The definitive scripts combined with a formal definition of the ADM provide a mathematical definition of the behaviour of the model (albeit subject to assumptions about interaction from the environment). This behaviour could, in principle, be recreated using other programming languages, such as C++ as used at the IBM WSDL.
- The LSD specification could be viewed as being a descriptive version of the Entity Relationship Diagram (ERD) used in SMOOA.
- The concepts of observable, agent and dependency correspond to concepts in SMOOA: observable to variable; agent to object; dependency to relationship between objects.
- Reconstructions of modelling processes tend to follow a pattern (LSD specification, visualization followed by animation) corresponding to SMOOA, SMOOD followed by coding.

However, it has since been recognized that these similarities are associated with a particular view of our modelling approach characterized by the absence of any consideration of the role of the modeller in the process. This absence is a direct consequence of focusing on existing models and reconstructions of the modelling process.

A view of EM that ignores the agency of the modeller effectively reduces it to what is termed an agentless or 0-agent system in EM, as shown in Example 2.3. In the absence of the modeller to explain what is meant by their model it is necessary for the model to use established, typically linguistic, conventions for representation. Certainly most LSD specifications, definitive scripts and accounts of modelling are written in such a way that they can be understood by those familiar with EM. These conventions must not be susceptible to change by individual agents. In general, a 0-agent system is one that, at some level of abstraction, does not exhibit change.



---

**Example 2.3. 0-agent view in OXO.** The modelling of the OXO game can be characterized as follows by ignoring the agency of the modeller and focusing instead on the artefacts produced and reconstructions of the modelling process:

- The definitions and actions of the board, player and umpire entities have an unambiguous behavioural interpretation within the framework of the ADM that could be implemented using conventional programming languages.
- The LSD board, player and umpire agents appear to specify the corresponding ADM entities.
- The observables, derivates and protocols of the board, player and umpire agents are given formal definitions in terms of the variables, definitions and actions of the corresponding ADM entities.
- Construction of the model follows an order (`geometry.e`, `display.e`, `status.e`, `sqvals.e`, `gamestate.e`, `control.e`).

Such a view of the modelling process gives it the character of a SD method. In fact, the definitive scripts generated during the modelling of the OXO game were used to implement a Pascal version of the OXO game.

---

### 2.1.3 Single-agent systems and modelling

Although the 0-agent view of EM is a valid interpretation it does not capture the essence of EM as experienced by the modeller. The use of the epithet *empirical* is meant to convey the central importance of the modeller's agency during the modelling process:

- The modeller correlates the experiences of the subject and the computer model through observation and experiment: the meaning of the computer model is defined by interaction [Bey97].
- The LSD specification records the observables, dependencies and agency as perceived by the modeller in the subject and represented in the computer model.
- The meaning of observables, agents and dependencies is given by the interaction of the modeller with the subject and model.

- The modeller is a free agent with actions determined by what the modeller knows, perceives and expects.

The activity of EM is an archetypal example of what is termed a 1-agent system in EM. Modelling involves the modeller changing the state of the model and subject by interacting with observables with a view to establishing a correspondence between the states, as shown in Example 2.4. In these cases the modeller is the sole instigator of state change within the modelling process. In general, a 1-agent system is a system in which one agent has the capacity to surprise.

---

**Example 2.4. 1-agent view in OXO.** By considering the experience of the modeller during the modelling of the game of OXO an altogether different view of the modelling process from that in Example 2.3 emerges. This 1-agent view of the modelling process is characterized by the following:

- The correspondence between the game of OXO and the computer model may change in unpredictable and surprising ways through interaction by the modeller, such as altering the shape of the board or the rules of play.
- The LSD description records the characteristic observables and dependencies of the board, player and umpire as identified by the modeller, such as the player seeing the board as lines and the umpire judging the next move based on the number of noughts and crosses.
- The meaning of observables and dependencies emerges through playing OXO and interacting with the model. For example, the observable `linesthru` is derived from the experiences of the modeller playing and representing the game of OXO.
- The order of model construction (`geometry.e`, `display.e`, `status.e`, `sqvals.e`, `gamestate.e`, `control.e`) reflects the way the modeller conceives the game of OXO.

This view of modelling the game of OXO captures the essence of EM. EM is used by the modeller to support their conceptualization of the game of OXO rather than as a method to represent preconceptions about the game.

---

This 1-agent approach to modelling is significant because models need only be understood by the modeller. In EM the objective is for the modeller to acquire an understanding of the subject themselves. This understanding does not require models that are understood by others, only models that can be seen as correspond-



ing to the subject by the modeller. Such models are subjective and personal in nature involving conventions whose meanings are dependent upon the modeller. A secondary objective in EM is typically to represent these more objectively using established conventions for representation so that they emerge within the 0-agent view of EM.

#### 2.1.4 Computer as artefact

It is the unusual status given to the computer in EM that allows the 1-agent approach to modelling. In EM the computer is only significant in so far as it serves as a physical instrument with which the modeller interacts. This is in contrast to the way in which the computer is conventionally regarded in classical computer science as a means to implement an abstract algorithm or computation. In effect, it is how the user perceives the computer as a physical object that matters in EM, not the invisible mechanism by which this object is specified [BNR95].

The status of the computer model in EM is similar to that of the spreadsheet. The only changes to the state of a spreadsheet are via actions on the part of the user. However, the essential spirit of EM is better represented where there is an explicit experiential aspect to the model. This could be achieved by the visualization of spreadsheet data. In EM the variables that appear in definitive scripts typically have an experiential significance - they may refer directly to entities visible to the computer user, such as points, lines, geometric attributes or windows on the screen for instance.

The EM tool that gives the computer the special quality that supports 1-agent modelling is the `tkeden` interpreter (Example 2.5). Almost all the models that have been developed using EM principles have been represented using the `tkeden` interpreter. This applies even to those that are constructed using the ADM, since this is at present implemented via a translator that acts as a front-end to `tkeden`, as shown in Figure 2.1.

A typical `tkeden` file comprises three kinds of construct: definitions, functions and actions. Definitions are formulated in terms of variables that represent scalar quantities, text strings and recursive non-homogeneous lists, as well as vi-



sually significant elements such as points, lines, and shapes in the form of planar line drawings, and windows in the screen layout. Functions serve as user-defined operators on the RHS of definitions; these supplement standard built-in operators that are used to define scalar, structural and geometric relations. Actions are specified as procedures that are triggered by changes to the values of particular variables [Bey97].

**Example 2.5. Computer as artefact in OXO.** Figure 2.2 shows a screen-shot of the EM tkeden during the modelling of the OXO game.

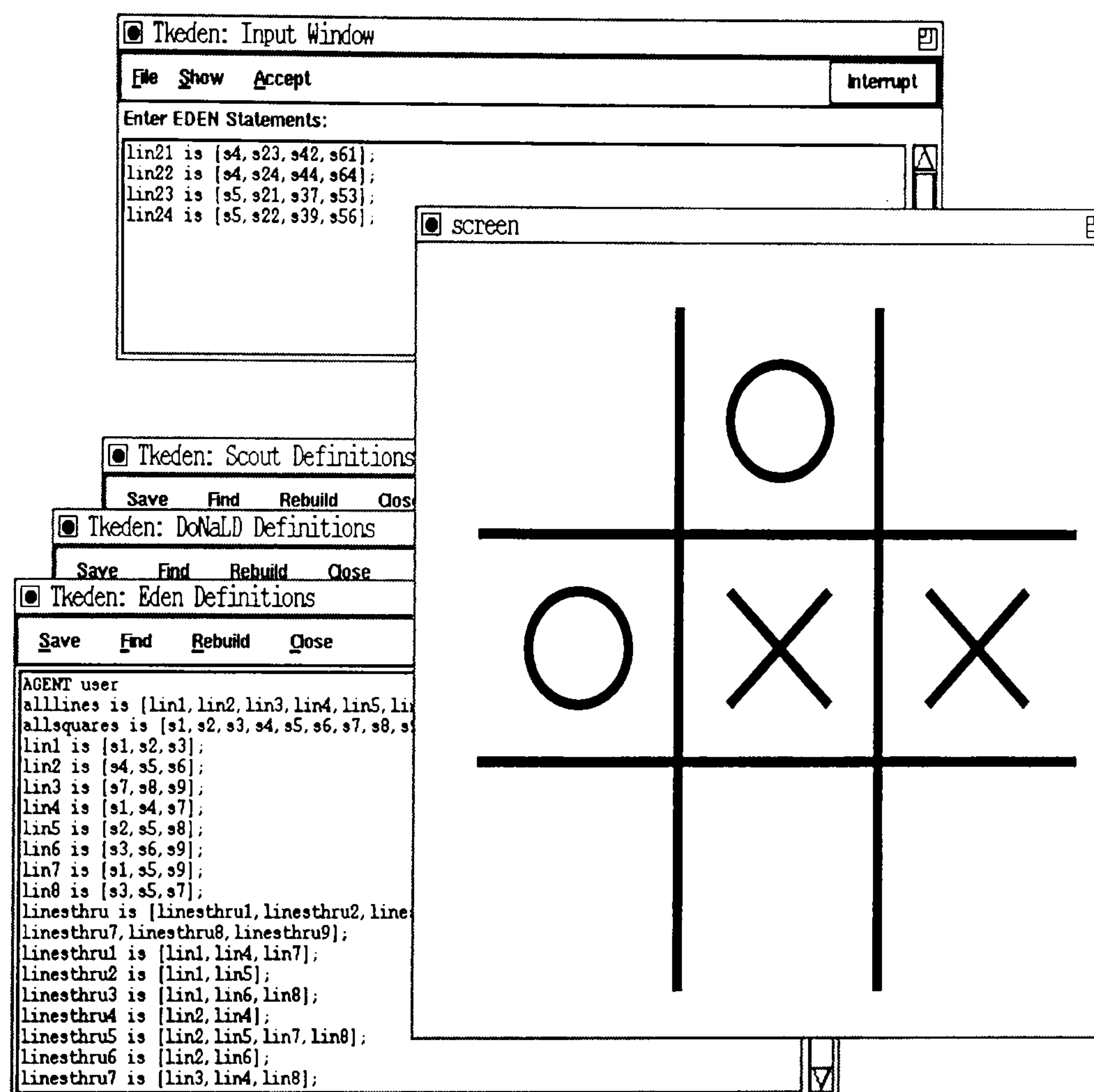


Figure 2.2: tkeden being used to model OXO game.

There are separate windows showing the EDEN, DoNaLD and SCOUT definitions. There is a window that shows the current state of the visualization. The tkeden input window is used to enter redefinitions of the scripts on-the-fly (the redefinitions shown are some of those being entered by the modeller to change the geometry of the board for 3-dimensional OXO).

The experiential character of 1-agent models in EM has crucial significance in respect to the relationship between computer model and subject. The focus on observables and dependencies in EM, when combined with mechanisms that

make these directly perceptible in interaction, is the means of ensuring that what is observed of the artefact (albeit as in caricature rather than as in realism) conforms to what is observed in the subject [Bey97].

### 2.1.5 Multi-agent systems and modelling

Although EM is essentially about the modelling of a subject by a single modeller using a computer it is often within the context of other human participants, as discussed in Example 2.6. The advantage of 1-agent modelling is that the modeller need not concern himself with how his model is understood by others. However, for most applications there comes a stage in modelling when the modeller has to communicate his use of artefacts and perhaps justify his actions to others. In general, a system with more than one agent (human or otherwise) is termed an n-agent system in EM.

By considering EM as a group activity it is possible to identify and associate roles with the human agents involved. The following are typical of the roles observed during modelling:

- the modeller who constructs the computer model;
- those familiar with the subject domain or who have a stake in the modelling;
- the objective or external observer whose view of the subject combines those of other agents.

Each of these roles might be fulfilled by different people in EM or a single person might have a number of roles. When more than one person is involved in EM communication becomes an important issue. In 1-agent EM the modeller typically assumes multiple roles in which communication is not an issue.

The EM concept of n-agent systems extends beyond social systems to systems including non-human agents. By associating human-like characteristics with the concept of agent the modeller assumes the roles of inanimate objects. This conceptual mechanism for assisting in the modeller projecting their own general characteristics onto the subject being modelled is an essential principle of EM.



---

**Example 2.6. n-agent view in OXO.** As was mentioned in Example 2.1 the modeller identified four agents in the game of OXO:

- the board;
- the player who places Xs on the board;
- the player who places Os on the board;
- the umpire who decides who plays next and who has won.

During the modelling of the game of OXO the modeller assumes the roles of the agents. In this way 1-agent modelling can be used to model n-agent systems.

The modeller adopts two different perspectives on the game of OXO during modelling:

- the game of OXO from the viewpoint of the board, player and umpire agents, and
- the game of OXO from the viewpoint of an external observer who sees the corporate effect of the board, player and umpire agents interacting.

The individual viewpoints are specified in LSD and the external viewpoint is represented by animating the agents within the ADM framework.

There comes a stage in modelling the OXO game when the modeller will want to test his model by letting others interact with it. By letting others interact with the model the modeller is testing his beliefs about the game of OXO and giving integrity to the model. This transforms the modelling of the OXO game from 1-agent to n-agent.

---

There are essentially two different ways in which EM can be applied to modelling systems of multiple agents [Bey97]:

- **Scenario 1** The modelling activity is centred around an external observer who can examine the system behaviour, but has to identify the components agents and infer or construct profiles for their interaction;
- **Scenario 2** The system can be observed from the perspective of its component agents, but an objective viewpoint or mode of observation to account for the corporate effect of their interaction has to be identified.

In many applications it is appropriate to consider both scenarios concurrently, with a view to reconciling global and local perspectives on the behaviour of a system.



It is in connection with systems with more than one agent or observer that LSD descriptions become significant. Modelling of n-agent systems can be viewed as involving two complementary principles that constitute concurrent engineering in EM [ABCY94c, ABCY94a, ABCY94b]:

- specifying agents in LSD by considering them in isolation;
- introducing a context for interaction by animating agents using ADM.

Arguably modelling n-agent systems is made difficult by the complex interaction between agents able to change the same observables. The modeller describes agents in LSD without having to address which observables are shared. Such issues are addressed in the ADM when the synchronization of interaction between entities is important.

### 2.1.6 Agent concept

The status of the modeller in 1-agent modelling is central to EM therefore the most appropriate way to conceive other agents is as having the same general characteristics of the modeller.

The characteristics of the modeller can vary, resulting in a broad interpretation of agency in EM. This broad spectrum of agency is categorized according to different views of an agent [Bey97], as illustrated in Example 2.7:

- **View 1** An entity comprising a group of observables with unexplored potential to affect system behaviour;
- **View 2** A View 1 agent that is capable of particular patterns of stimulus-response within the system.
- **View 3** A View 2 agent whose pattern of stimulus-response interaction can be entirely circumscribed and predicted.

In EM, each of these views has a different status, and there is a tendency to progress from the first to last view of an agent during modelling. EM is of interest somewhere between View 1, where the agent concept is vacuously broad, and View 3, where it is

impotent. This interest centres around our uncertainty about the status of entities in respect to agency. In View 1 our concern is whether an entity has any influence and in View 3 our concern is whether the exact nature of its influence is known.

The classification of agency according to these views is not a formal matter. Agency is being invoked as a conceptual device fundamentally associated with how phenomena are construed to occur. EM promotes the view that agency is only meaningful in relation to the development of understanding.

---

**Example 2.7. Classification of agents in OXO.** The agents in the OXO game and its representation show the full range of agency accommodated in EM:

- The opponent at the start of modelling and the board are examples of a View 1 agent with unexplored potential for affecting the game of OXO.
- The opponent, once the player-modeller has identified it as having a similar role to themselves, is an example of a View 2 agent with identified patterns of stimulus-response in the game of OXO.
- An automatic player entity is an example of a View 3 agent whose pattern of stimulus-response is made entirely predictable by the framework of the ADM.

Although all these views of agent appear within the modelling process the focus of attention is on the View 2 player agents. EM involves the modeller moving from a View 1 through to a View 3 of agents.

---

### 2.1.7 Conceptualization

EM can be thought of as the means by which the modeller represents the conception of the subject as it evolves [Bey97]:

1. Interaction with artefacts: identification of persistent features and contexts.
2. Practical knowledge: correlations between artefacts, acquisition of skills.
3. Identification of dependencies and postulation of independent agency.
4. Identification of generic patterns of interaction and stimulus-response mechanisms.
5. Non-verbal communication through interaction in a common environment.



6. Phenomenological uses of language.
7. Identification of common experience and objective knowledge.
8. Symbolic representations and formal languages: public conventions for interpretation.

The stages of EM represent a progression from a subjective to an objective view of the subject, as in Example 2.8. The early stages correspond to the modeller's view of the subject during 1-agent modelling. In later stages the modeller develops methods of communication as typified in n-agent modelling. Finally the model acquires a meaning independent of the subject and modeller (0-agent system).

---

**Example 2.8. Conceptualization in OXO.** It is possible to match the stages in constructing the model of the OXO game given in Example 2.2 with the stages of conceptualization given in Section 2.1.7:

- Construction of `geometry.e` (What is the geometry of the board?) and `display.e` (How does the actual board and what I perceive conform?) corresponds to interaction with artefacts: identification of persistent features and contexts.
- Construction of `status.e` (Can I interpret the board in OXO terms?) and `sqvals.e` (What considerations guide me in contemplating the next move?) corresponds to practical knowledge: correlations between artefacts, acquisition of skills.
- Construction of `gamestate.e` (Whose turn is it to play?) corresponds to
  - identification of dependencies and postulation of independent agency;
  - identification of generic patterns of interaction and stimulus-response mechanisms;
  - non-verbal communication through interaction in a common environment;
  - situated use of language;
  - identification of common experience and objective knowledge.
- Construction of `control.e` corresponds to symbolic representations and formal languages: public conventions for interpretation.

The construction of the model of the OXO game reflects the conceptualization of the OXO game by the modeller.

---



It is this natural progression from the subjective to objective view of a system that provides the EM activity with its order rather than the modeller following a prescribed method. The aim of EM is to support the natural process of conceptualization rather than prescribe essentially unnatural methods of analysis.

### 2.1.8 Situating EM

The closest conventional computing comes to EM is in the use of spreadsheets. Using a spreadsheet illustrates agency in a 1-agent system. The semantically interesting state is in the relationship between the states of the spreadsheet and the part of the real-world it models. The only significant changes to the state are via actions on the part of the user. In [Nar93b] Nardi presents a particularly interesting study of the impact of spreadsheet use on the SD culture. The themes emerging from this study - support for interaction, re-use and extensibility - are consistent with our experience and aspirations for EM.

The distinction between EM approaches and formal approaches to describing behaviour in computer science is highlighted by Brian Cantwell Smith [Smi95, Smi87]. Smith distinguishes between the semantics of a program as it is understood in theoretical computer science and the relationship between these semantics and the external world. The real-world meaning of a program, for which EM provides a means of development [Bey92], is appropriately termed the “the semantics of the semantics” of programs by Smith. As Smith’s analysis makes clear, knowing the semantics of a program and knowing how to deal with the semantics of the semantics of a program are quite different issues.

As the title “Empirical Modelling” suggests, our approach to modelling is rooted more in the philosophy of empiricism than rationalism and logic. The work of the American philosopher William James [Jam96] indicates that “Radical Empiricism”, rather than traditional empiricism, provides the more appropriate philosophical foundation to our modelling method. James argues that by identifying sensory particulars the traditional empiricists break up the “conjunctive relations” that are “pure experience”: “Conception disintegrates experience utterly” ([Jam96] p70), “[it] performs on conjunctive relations the usual rationalistic act of substitution -



[taking] them not as they are given in their first intention, as parts constitutive of experience's living flow, but only as they appear in retrospect, each fixed as a determinate object of conception, static, therefore, and contained within itself." ([Jam96] p236).

This philosophical outlook of James can be recognized in Gooding's account of scientific discovery and Faraday's discovery of electromagnetism. In his book [Goo90] Gooding rejects the conventional approach of analyzing workbooks and reconstructing methods. Instead, Gooding presents the notion of a "construal" that is "a means of interpreting unfamiliar experience and communicating one's trial interpretations" and uses this as a vehicle for understanding how scientists conceive and communicate their understanding of novel phenomena. There are parallels between the role of the construal in experimental sciences and the computer model in EM and the nature of the process of constructing such artefacts.

Understanding the behaviour of modellers inevitably leads to psychological considerations, in particular cognitive psychology because of the interest in understanding how modellers come to know about the subject during EM. As far as EM is concerned perhaps the most interesting explanations of cognitive processes are those that involve the creation of a model of the world in the mind, such as mental modelling proposed by Johnson-Laird [JL83]. This suggests that in constructing a computer model the modeller is mirroring and supporting mental processes. Other psychologists have tried to identify the qualities of models that support the conceptualization of novel systems [FWS92].

## 2.2 Product design

In this thesis PD means product design in the sense of Pugh's vision of *total design*: "the systematic activity necessary from the identification of the user need to the selling of the successful product to satisfy the need - an activity that encompasses product, process, people and organization" [Pug91]. His model of total design is meant as a framework for what design is rather than a prescriptive method of how design should be done. Pugh's view of design has been adopted as the basis of learning design in over 80 institutions within the United Kingdom. The books



“Total Design: Integrated Methods for Successful Product Engineering” [Pug91] and “Creating Innovative Products Using Total Design: the Living Legacy of Stuart Pugh” [Pug96] describe total design and are reviewed in Appendix D.

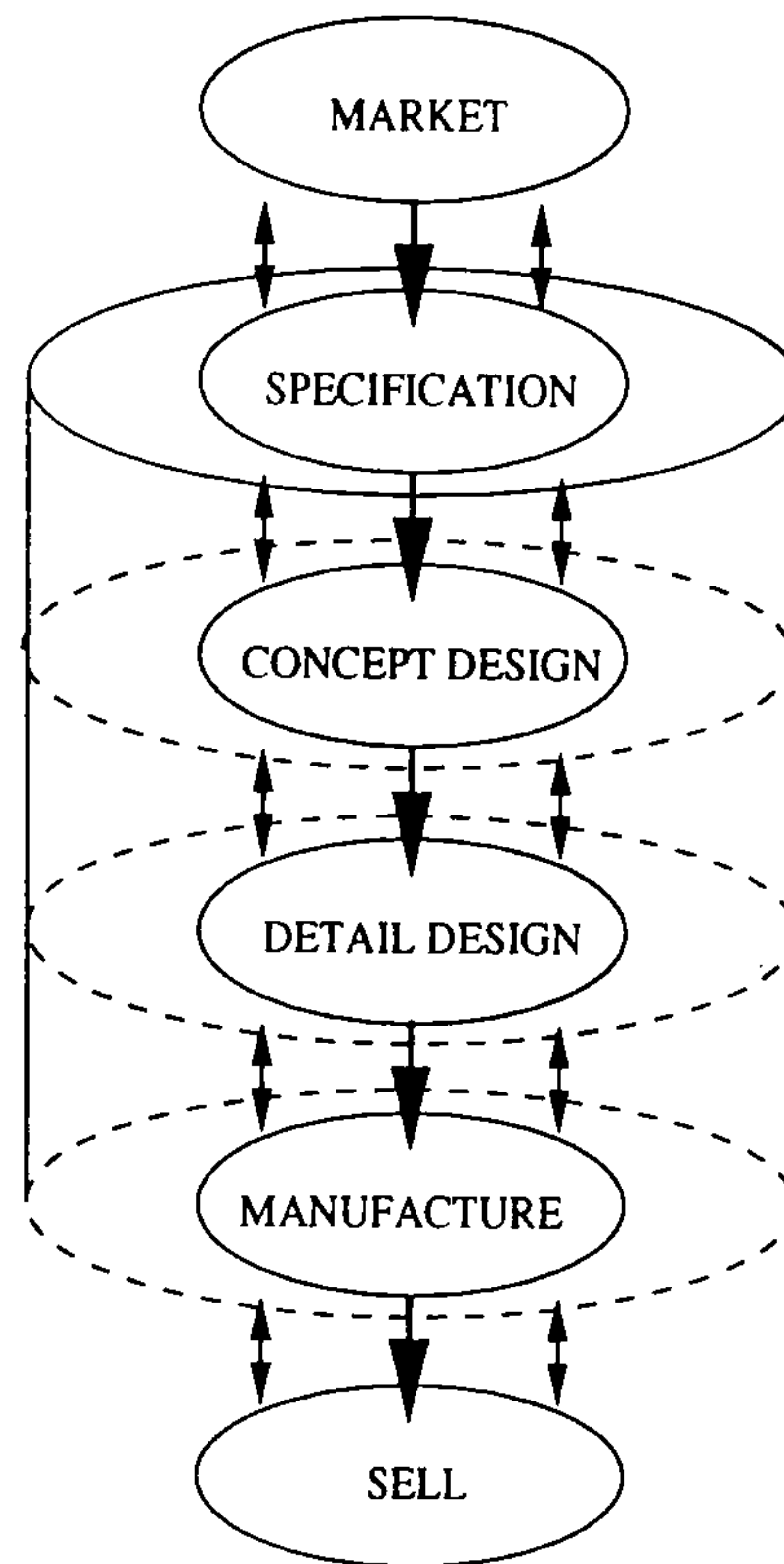


Figure 2.3: Activity model for total design.

The model of total design, shown in Figure 2.3, has a central core of activities all of which are imperative for any design irrespective of domain. The design core consists of the investigation of the market, development of a product design specification, conceptual design, detail design, manufacture and selling of the product. Design starts with a need that, when satisfied, results in a product that fits into an existing market or creates a market of its own. From the statement of the need a *product design specification (PDS)* is formulated which is the specification of the product to be designed. The PDS acts to constrain the total design activity by placing boundaries on the stages in the design core. Other constraints that are specific to particular designs, such as management, quality, information, techniques and technology, are discussed in Section 6.7 with respect to management and quality in EM and product design.

Although all the stages of total design are of equal importance the stages of specification, conceptual design and detail design are detailed below because they set the context for the discussion of design in this thesis:

- The starting point for any design activity is an investigation of the market from which a comprehensive PDS is prepared during the *specification* stage. Pugh makes it clear that the PDS is the specification of the *product to be designed*, not the specification of the product itself. The latter manifests itself only on completion of the design activity.

At the end of the design activity the design of the product should fit the PDS that may have changed along the way. If during the design of a product there is good reason for changing the PDS then it is changed. It is considered by Pugh as an evolutionary, comprehensively written document which upon completion of the design activity has itself evolved to match the characteristics of the final product.

- The *conceptual design* stage is primarily concerned with the generation of solutions to meet the PDS. In fact, this stage combines the generation of solutions to meet the PDS with the evaluation of solutions to select the ones that best fit the PDS.

In generating solutions the designer must come up with concepts which he believes fit the PDS and communicate these ideas for evaluation. Central to this conceptualization and representation is the process of synthesis. The designer mentally synthesizes familiar images and concepts from his knowledge and experience, with the PDS in mind, generating concepts for the system as a whole. The designer's ideas are represented as sketches, models, documents and prototypes for the purpose of evaluation.

In evaluation choices have to be made about which solutions to reject and which solutions to keep for further refinement. Pugh argues that optimization provides a partial solution to the problem of evaluation because of its reliance on quantifiable evaluation criteria based on the PDS. Pugh suggests a total solution to the problem of evaluation based on decision matrices that he calls the method of *controlled convergence*: with concept names heading the rows and evaluation criteria heading the columns the design group comes to a consensus on scores to complete the matrix. It is expected that completing the



matrix will lead to the emergence of new solutions to evaluate.

- Components and sub-systems are engineered in the *detail design* stage. During the conceptual stage of design the designer becomes increasingly involved in the detail design of the concept. The focus of design moves from the design as a whole to individual subsystems and components.

By the end of the conceptual design stage the designer has detailed knowledge of the properties needed of components. This knowledge is stated in the form of a component design specification (CDS). The CDS is simpler than the PDS with a shift of emphasis. It is simpler because many of the criteria such as testing, packing, shipping, aesthetics and ergonomics are not relevant at component level. However, the performance, which is essentially the behaviour of the component, is important at the component level.

Pugh points out that, although the sequence of these stages is typically ordered as specification, conceptual design then detail design, the “design flow” is bidirectional between stages. Detail design can influence conceptual design which can in turn influence the specification of the product.

## 2.3 Software development

In this thesis SD means mainstream software development as exemplified by the Shlaer-Mellor object-oriented analysis and design method adopted at the IBM WSDL (Appendix B). Such approaches are characterized by an object-oriented analysis method for transforming the requirements for a system into code.

SD is traditionally divided into the stages of analysis, design, coding and testing or maintenance [Roy70, You92]. Originally meant to be in strict sequence [Roy70], more recent versions of the model [Boe85] show stages repeating and bidirectional flow between stages, with perhaps the most radical being the prototyping lifecycle [Boa84]. Although all of the stages are essential to SD this thesis concentrates on the stage of analysis. Analysis is the examination and representation of a real-world system for the purpose of designing and implementing software. The



products of analysis typically determine what system is to be designed and implemented making it generally recognized as the most important stage in SD.

Most recently the discipline of requirements engineering [LK95, Lam88, Hof93, Poh96] has become associated with systems analysis [Gog94, SS96]. Requirements engineering is “the systematic process of developing requirements through an iterative co-operative process of analysing the problem, documenting the resulting observations in a variety of representation formats and checking the accuracy of the understanding gained” [Poh96]. Requirements engineering results in the formulation of a precise description of the system known as a requirements specification or statement of requirements. Although it can take many forms, ranging from informal natural language to more formal graphical and mathematical notations [LK95], it is generally agreed [Lam88, Dav93, Hof93] that a statement of requirements should be, or aim to be, complete, correct, unambiguous, understandable, modifiable and consistent [DT90] with respect to the system.<sup>1</sup>

Throughout the brief history of SD there have been many methods proposed for performing analysis [DeM78, Jac83, SM88, CY90, Mar90, WBWW90, R<sup>+</sup>91, Rum93, Jac92, SM92, Boo93, Boo86, YC75, Mey88]. Today, the most popular methods for analysis are object-oriented methods [You92] based on the notion of an Object. The object-oriented Object concept (starting with a capital to distinguish it from the word object that has a different meaning as discussed in Section 6.4.3) [Boo93, CY90, Mey88, Nie89] is a mechanism for abstraction and generalization. An Object consists of an interface and implementation. The interface is an abstraction of the implementation and the only part of the Object concept that is visible to clients of the Object. The implementation which provides the functionality defined in the interface is hidden. Objects with the same interfaces are classified together. The Object class definition contains the names of the service actions provided by the Object implementation. The class definition also defines the structure of the Object class in terms of other classes. In this way the Object concept deals with the representation, organization and abstraction of the structural, behavioural and func-

---

<sup>1</sup>This account is meant to represent the current status of requirements engineering. However, requirements is perhaps the most rapidly evolving field within SD. Section 6.5 discusses how the future of requirements engineering relates to EM.



tional aspects described in the statement of requirements. Object-oriented methods of analysis focus on the structure, behaviour and functionality of the system.

There are many methods of object-oriented analysis to choose from [Nie89, WBJ90, MP92b, FK92]. However, although each has different notations, their underlying concepts and principles are very similar. In this thesis a hybrid method is used that is based on the notations, concepts and principles of the established object-oriented analysis methods of Coad-Yourdon [CY90], Shlaer-Mellor [SM88, SM92, Lan93, FHRK93] and Rumbaugh [Jac92]. This hybrid is intended to highlight the essential nature of object-oriented approaches by stripping away the largely idiosyncratic stylistic complexities of specific notations [FS97, BRJ98b, BRJ98a].

Object-oriented analysis involves constructing separate models of the structure, behaviour and function of a system. The models are constructed in order, with the information given in each model being used in the construction of subsequent models. The models and order of development are as follows:

- The *structure model*, shown in Example 2.9, represents the organization of the system into Object classes. Labelled boxes represent Object classes. Labelled arrows between boxes represent structural, such as *landing button is a button* (inheritance), and functional associations between Object classes, such as *shaft operates brake*. Functional associations correspond to actions performed by Objects. Single and double headed arrows are used to indicate how instances of Object classes are associated with one another.
- The *behaviour model* or state model, shown in Example 2.10, represents each Object class lifecycles as a state-transition diagram. The states are represented by labelled boxes. The state transitions are represented as directed arrows, each labelled with an action and event name. For example, the event *applying* results in the action *apply*. The actions are the services the Object class provides. Action names label the heads of arrows, representing transitions, to indicate that it is to be executed when the next state is entered. The transition is made when the named event is generated by an action.

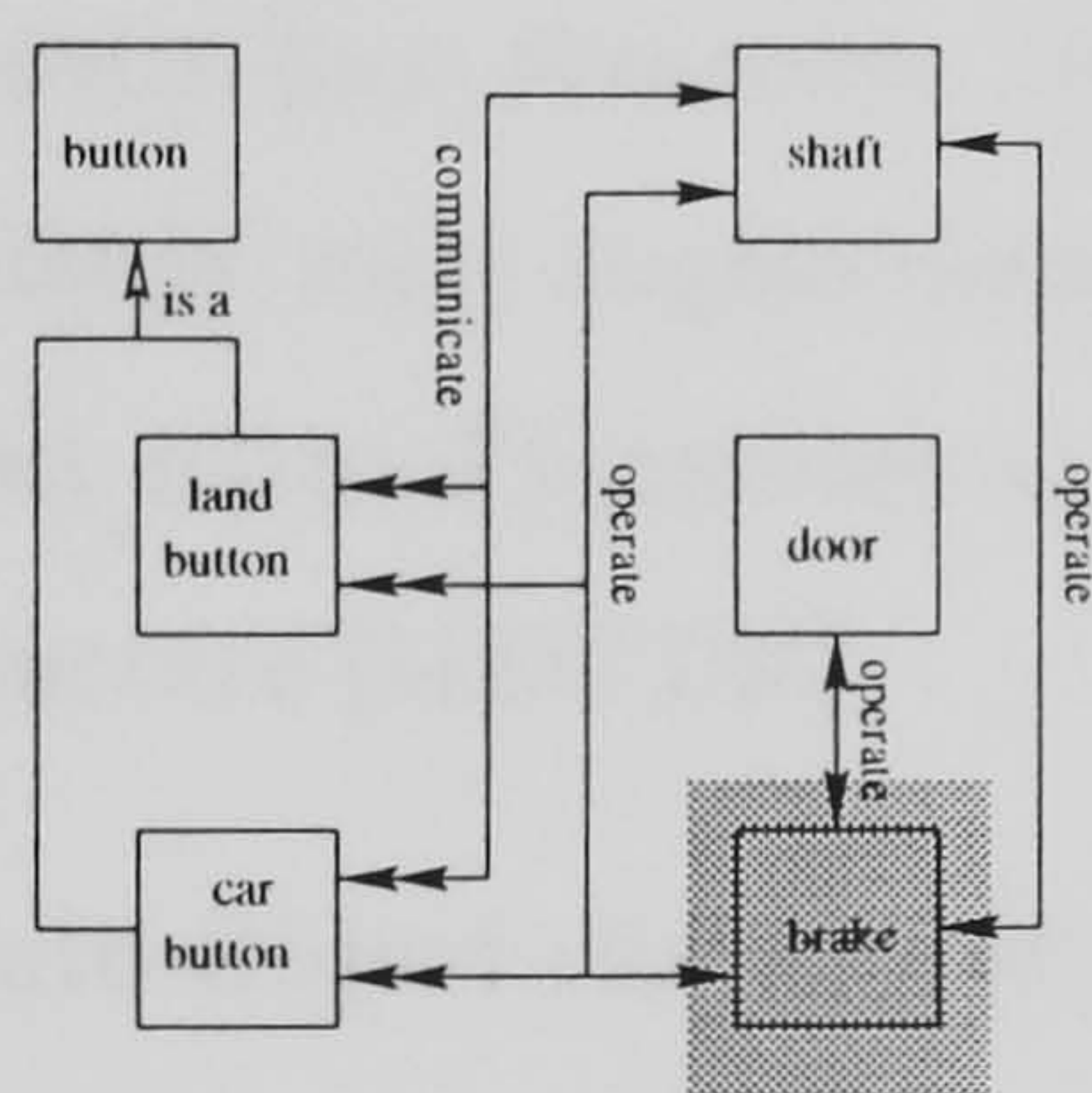
Actions are represented by sequences of instructions. The sequences are typi-



cally short and simple because there are many actions distributed among the Object classes.

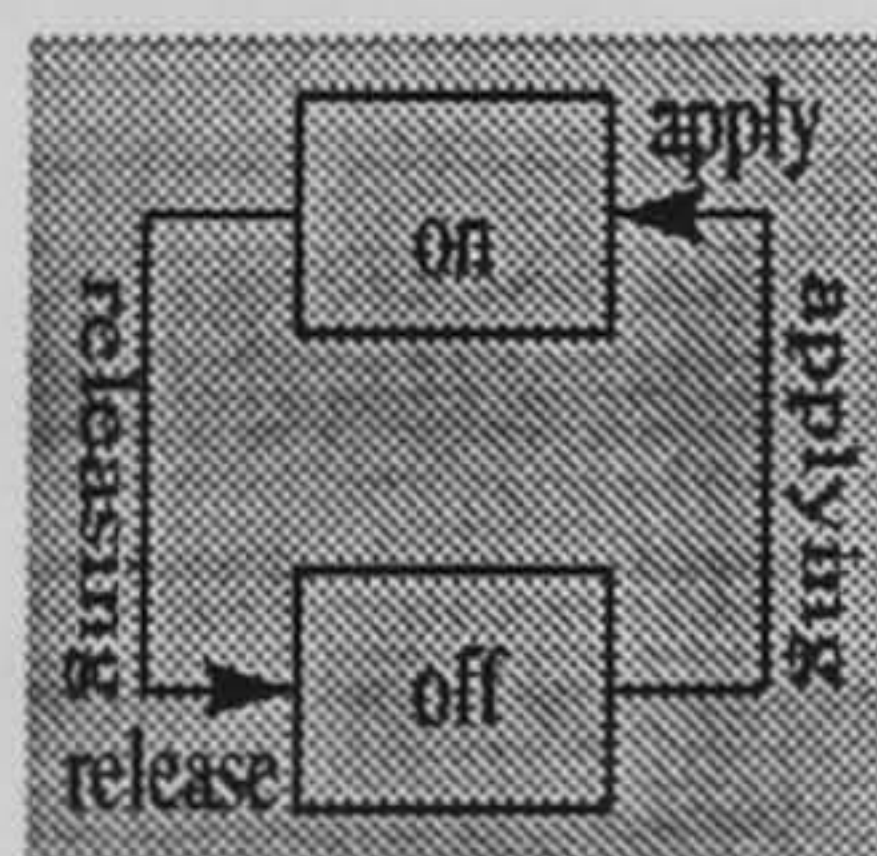
- The *process model* or function model, shown in Example 2.11, represents the system as a process. Each action is represented by an oval labelled with the action name. If the execution of the action results in the generation of an event then a directed arrow is drawn between the oval representing the action and any ovals representing actions which would be executed as a result of a state transition. Data stores used by actions are represented by parallel lines labelled by the variable name.

**Example 2.9. Structure models in SD.** The structure model for the MUL lift



shows the part of the model corresponding to the brake mechanism as highlighted.

**Example 2.10. Behaviour models in SD.** The behaviour model and action definitions for the MUL brake



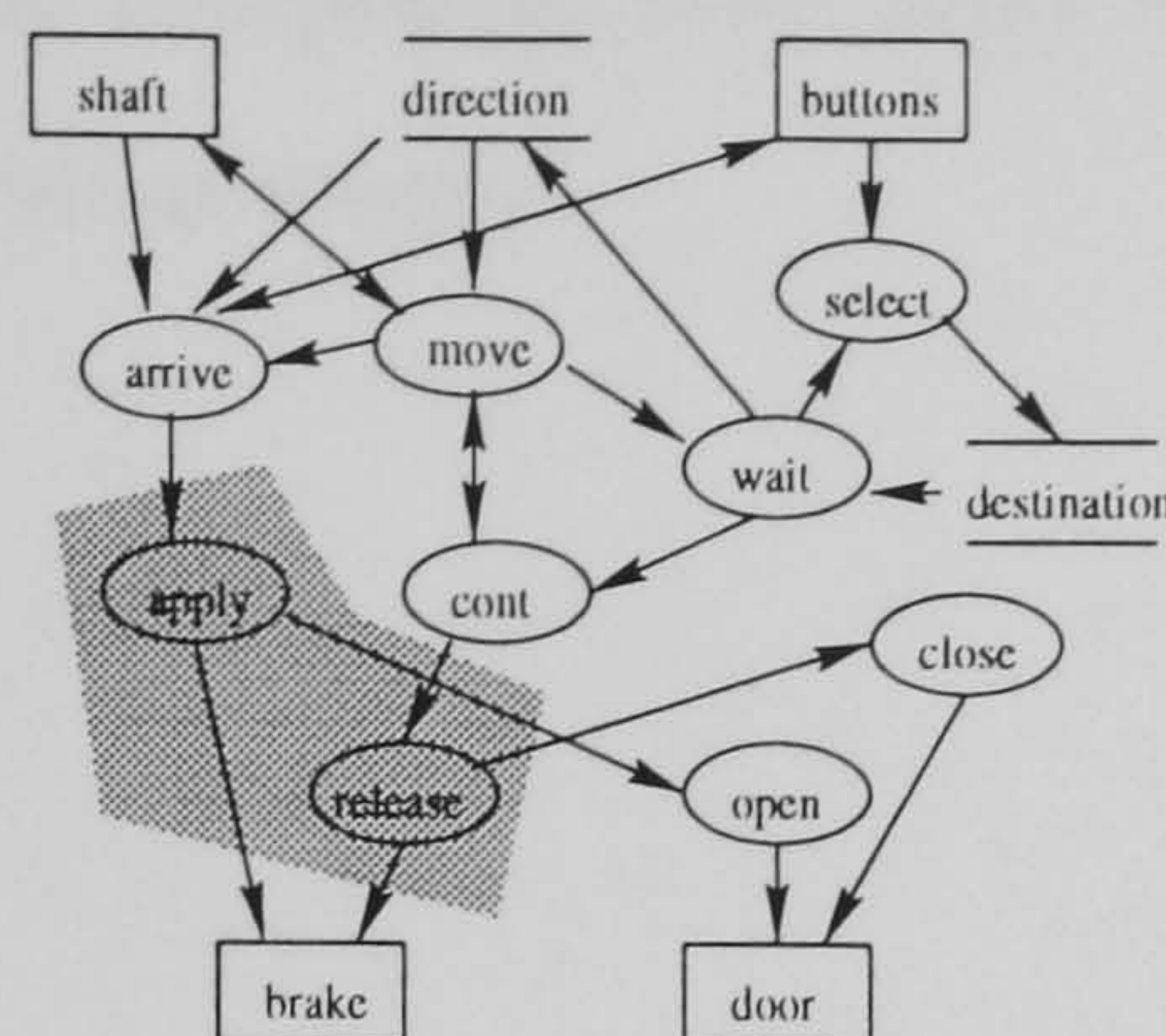
```
apply is
    apply brake;
    generate opening.

release is
    generate closing;
    release brake
```

is typical of the models defining the behaviour of the shaft, door and button classes.



**Example 2.11. Process models in SD.** The process model for the MUL lift



shows the part of the model corresponding to the brake mechanism as highlighted.

Typically, the construction of the SD models is based on a statement of requirements. In such cases, the software developer uses conventions for transforming the statement of requirements into structure, behaviour and process models. Such a transformation particularly suits requirements stated in formal languages [Bac87, C90, M<sup>+</sup>88, Dro89] but natural language requirements can also be transformed based on its logical structure [MEGT96]:

- Nouns are transformed into Object classes and attributes.
- Noun phrases are transformed into structural associations between Object classes.
- Verbs are transformed into actions.
- Verbs phrases are transformed into functional associations between Object classes, transitions between states and data-flows between actions.

Such an approach places emphasis on the description of the abstract notions of structure and function, represented within the surface structure of the statement, as opposed to more concrete concepts embodied within the meaning of nouns and verbs [Goo90, Who78].

Once the analysis of the system is complete the design of the software begins. This move from analysis to design is characterized by a shift from the problem domain of the real-world system to the solution domain, consisting of software components with which to build the required system [MP92b]. There are clearly parallels



between engineering design in PD [Pug91] and design in SD since both focus on the construction of systems at the component level after the organization of the parts has been established in a previous stage.



## Chapter 3

# Characterization of EM, PD and SD

An insight into how EM, PD and SD differ in character is achieved by comparing them. However, it is important to find an appropriate framework for comparison because the three activities are very different in nature. Consideration of artefacts forms a suitable basis for such a framework. The various aspects of EM, PD and software development are then compared with respect to their use of artefacts.

This chapter compares EM, PD and SD. The comparison focuses on the artefacts: the LSD specification, visualization and animation in EM, the sketch in PD, and the structure, behaviour and process models in SD. In addition to artefacts, the actions, subjects, constraints, environments and knowledge of the activities are compared: knowledge informs actions on an artefact within an environment to represent a subject under certain constraints.

### 3.1 Background

In the previous chapter EM, PD and SD were described as processes. Many activities are described in this way as processes comprising phases performed in sequence, with each phase characterized by the construction of an artefact. However, such a description is essentially a reconstruction of an activity and not a true characterization of how the activity happens [Goo90, Kap64]. A retrospective view of EM, PD

and SD tends to give a misleading account of a planned sequence of actions, when, in fact, the actions were determined by the situation at the time [Sim81].

The lift project case-study provided a particular context in which to consider EM (LSD specification, visualization and animation), PD (sketch) and SD (structure model, behaviour model and process model). It was found that the character of EM, PD and SD was determined by the nature of the artefacts involved. In effect, the nature of the activities was shaped by the particular kind of artefacts used by modellers, designers and software developers in representing the subject. The nature of the artefacts was discovered to influence many aspects of the activity:

- the subject of the activity;
- the actions of the activity;
- the constraints that limit the activity;
- the environment in which the activity happens;
- the knowledge to perform the activity.

Each of these aspects of EM, PD and SD is discussed in the remainder of this chapter.

## 3.2 Artefacts

It was discovered that particular views of the subject were clarified during the construction of the EM, PD and SD artefacts in the lift project. The modellers found that the view of the system as a collection of observables and agents was made clearer during the construction of the LSD specification. Construction of the other EM, PD and SD artefacts was found to clarify the view of the lift system as a structure and the view of the lift system as a structure with a purpose. The artefacts were found to represent the structure and function of the subject in fundamentally different ways. See Appendix C for all the artefacts constructed during the lift project, including associated DoNaLD and ADM scripts and statements of requirements.

The LSD specification, visualization and animation were the artefacts constructed by modellers during EM in the lift project. The LSD specification, shown



in Example 3.1, was introduced in the previous chapter. The term *visualization* is used in this thesis to mean the artefact resulting from `tkeden` interpreting a DoNaLD script. The visualization is essentially a line drawing that the modeller interacts with by changing the values of DoNaLD variables, as shown in Example 3.2. The visualization has no automatic behaviour - such behaviour is characteristic of the animation. The term *animation* is used in this thesis to mean the artefact resulting from `tkeden` interpreting a DoNaLD and ADM script. The ADM entities animate the visualization by changing the values of DoNaLD variables, as shown in Example 3.3.

The sketch was the artefact constructed by designers during PD in the lift project, as shown in Example 3.4. It seems to be widely accepted that the *sketch* is the principal artefact in the conceptual design phase of PD described in the previous chapter [Pug91, Pug96, Fer92]. This thesis concentrates on how the sketch was used by designers in the lift project as a means of turning their ideas for products into product designs with a view to subsequent detail design and manufacture [Pug91, Pug96].

Software developers constructed structure, behaviour and process models during SD in the lift project, as shown in Example 3.5. These artefacts seem typical of those produced during object-oriented analysis using the mainstream methods outlined in the previous chapter. The *structure model* represents the organization of the system into Object classes, the *behaviour model* represents the Object class lifecycles, and the *process model* represents the dataflows between Object classes. This thesis concentrates on how the models were used by software developers in the lift project as a means of capturing the structural and functional aspects of the subject with a view to subsequently designing and coding software.

**Example 3.1. LSD specifications in EM.** The LSD specification of the Hydrolift pump agent

```

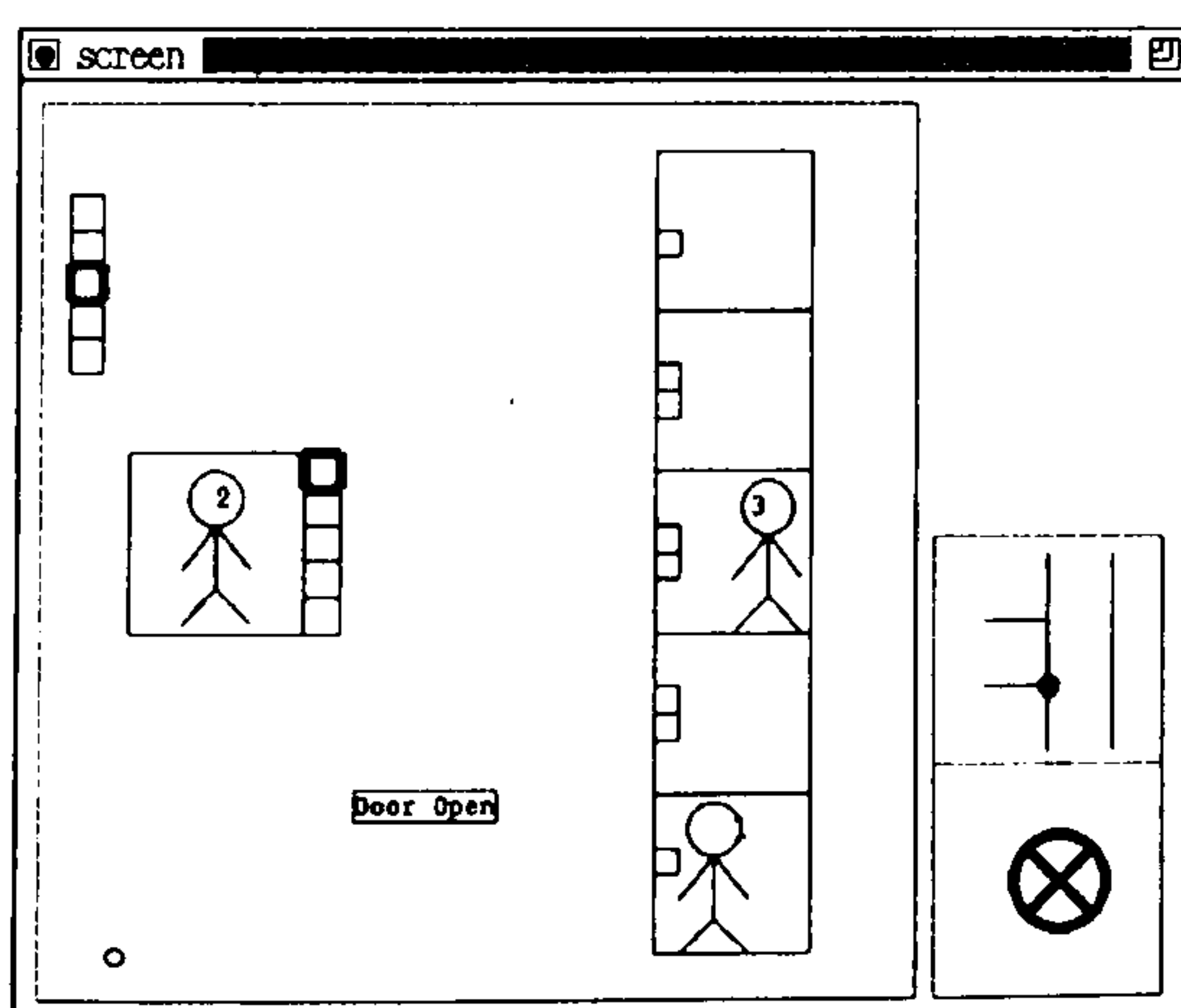
agent pump() {
state
  change target
oracle
  brake pressure chan1
handle
  brake pressure chan2
derivate
  k = 100,
  change is (pressure < target) ? k :
            (pressure > target) ? -k : 0
protocol
  target == pressure + change && brake == OFF
                                -> brake = ON,

  change == 0 -> target = chan1*k,
  pressure == target -> chan2 = target/k,
  brake == OFF -> pressure = pressure + change,
  brake == ON && change != 0 -> brake = OFF
}

```

shows how agents in an LSD specification are defined in terms of observables (states, oracles and handles), derivates and protocols.

**Example 3.2. Visualizations in EM.** The screen-shot shows the state of the visualization of the Hydrolift after redefinitions (a) to (h). The sequence of redefinitions changes the state of the visualization to mimic a user on floor 2 travelling to floor 5.

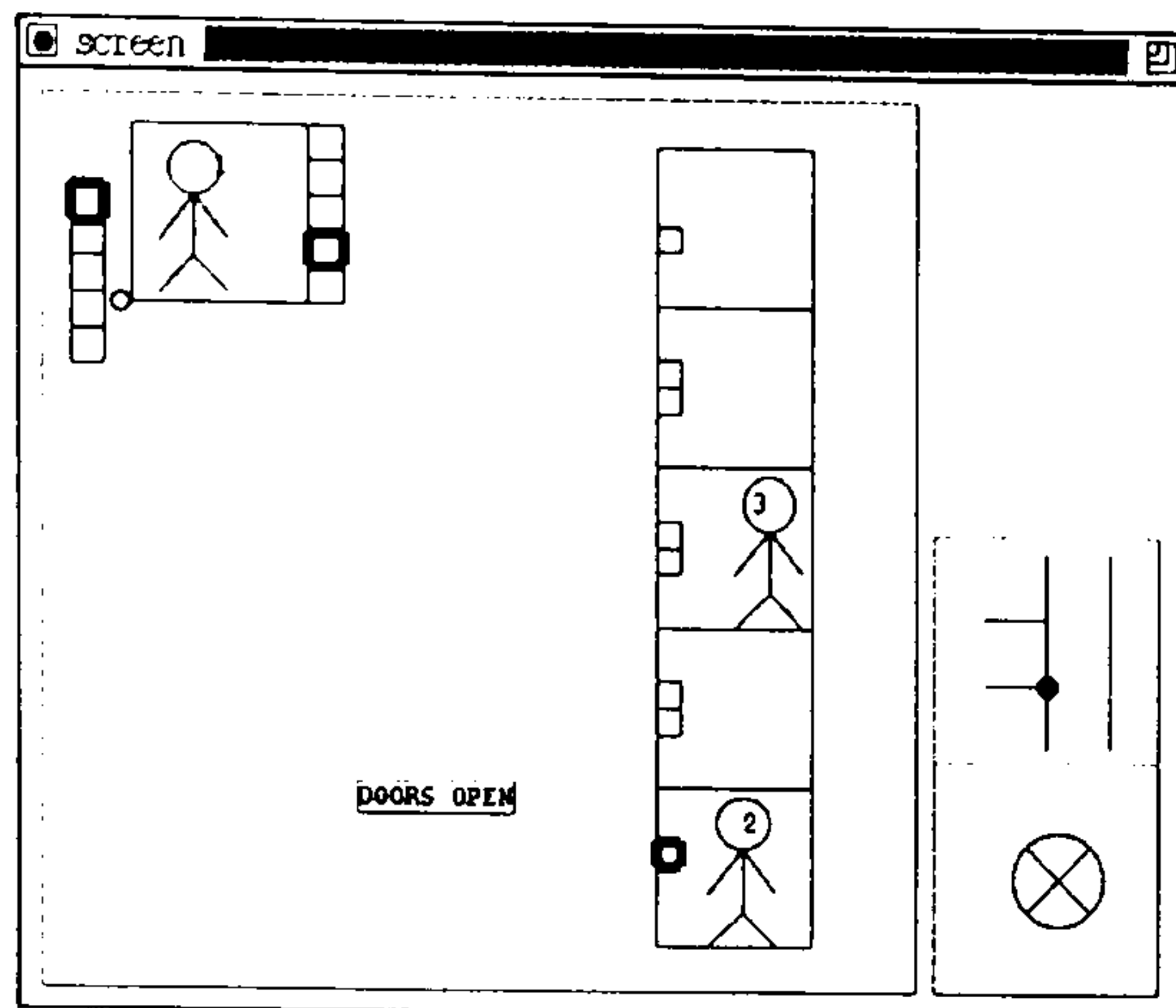


liftfloor = 2	# a car at floor 2
dooropen = true	# b door open
inliftB = true	# c userB enters car
floorB = liftfloor	# d link user-car
car/button5/light = true	# e select floor 5
dooropen = false	# f door closed
pumpshape/on = true	# g pump on
liftfloor = 3	# h floor 3
liftfloor = 4	# i floor 4
liftfloor = 5	# j floor 5
car/button5/light = false	# k car arrived
pumpshape/on = false	# l pump off
dooropen = true	# m door open
inliftB = false	# n userN leaves car
floorB = 5	# o unlink user-car

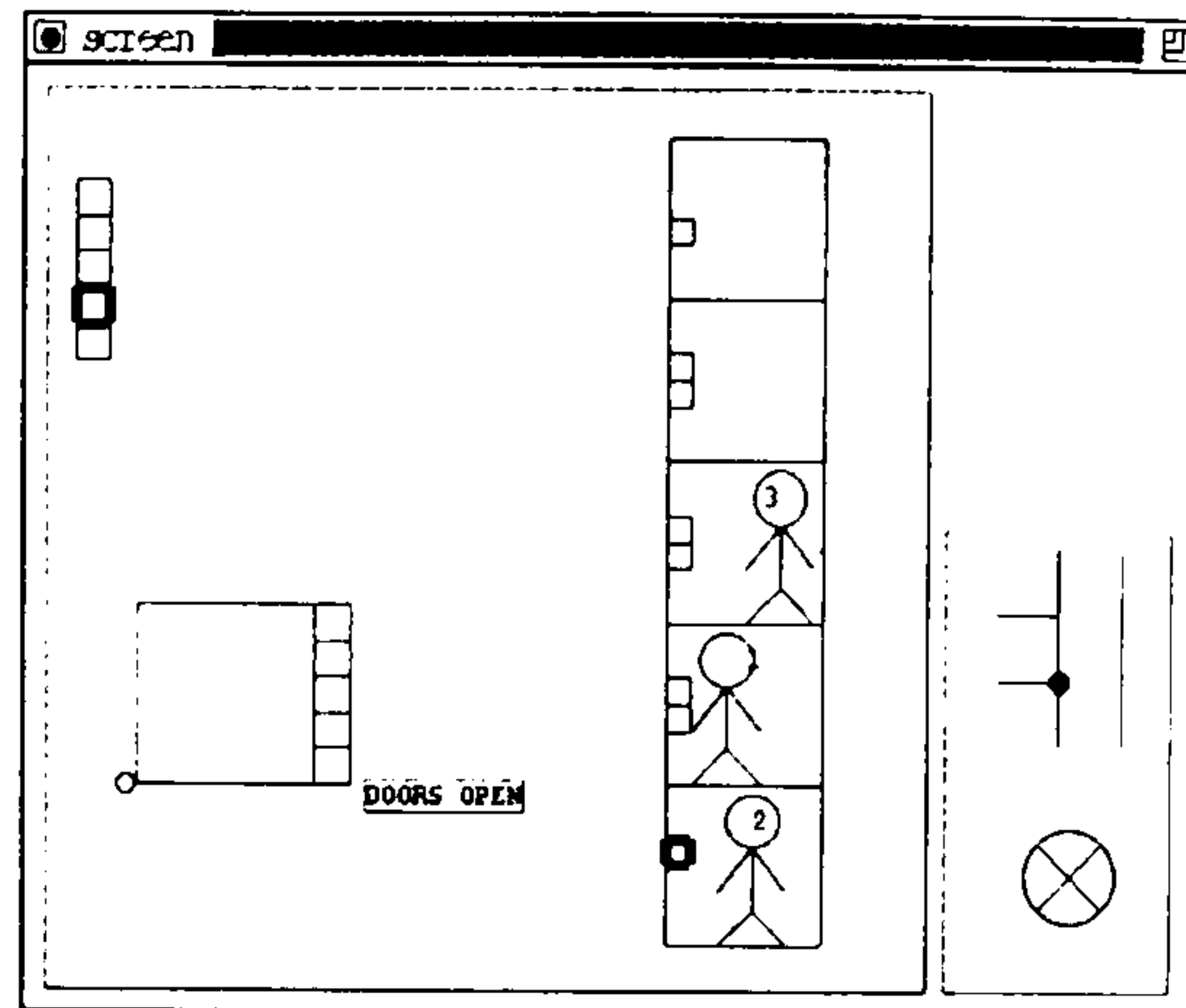
Notice that in (d) variables are synchronized in change and in (e) and (k) the hierarchical structure of the car and button shapes is used.



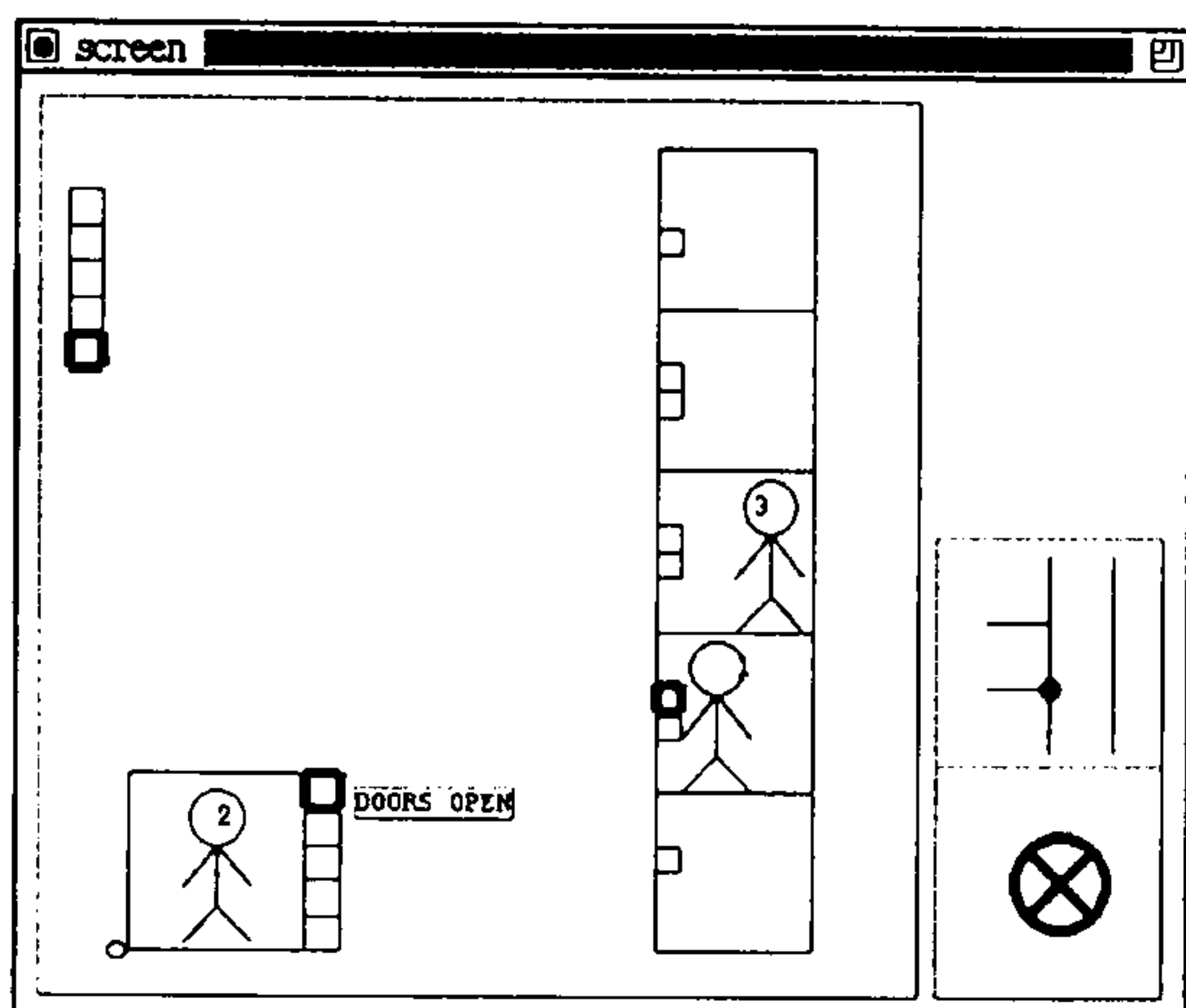
**Example 3.3. Animations in EM.** The screen-shots show the visualization shown in Example 3.2 with the addition of ADM entities for each user and the lift mechanism providing an automatic behaviour.



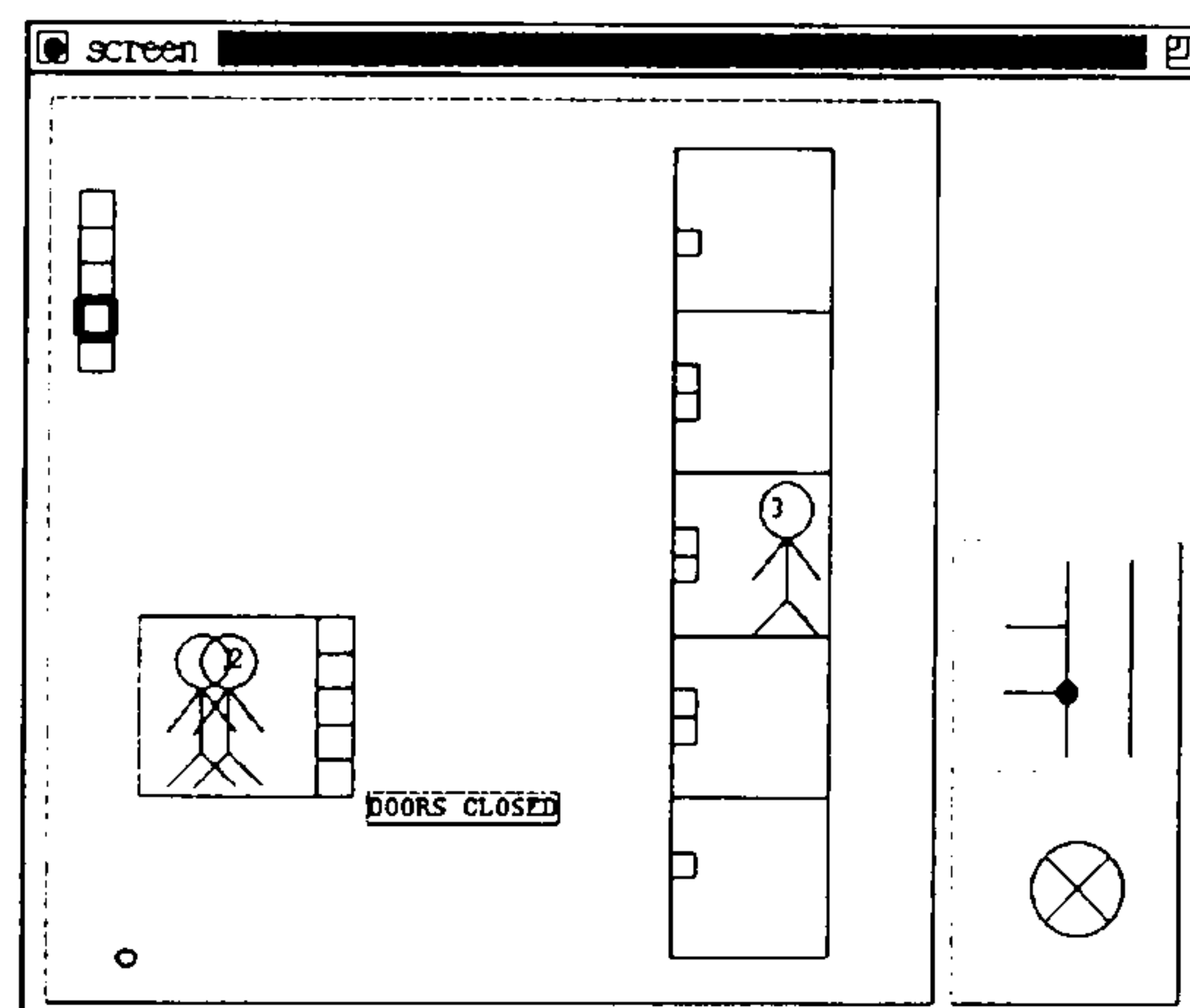
(a) User 1 enters and requests floor 2.



(b) Arrives at floor 2 and user 1 exits.



(c) User 2 enters and requests floor 5 .



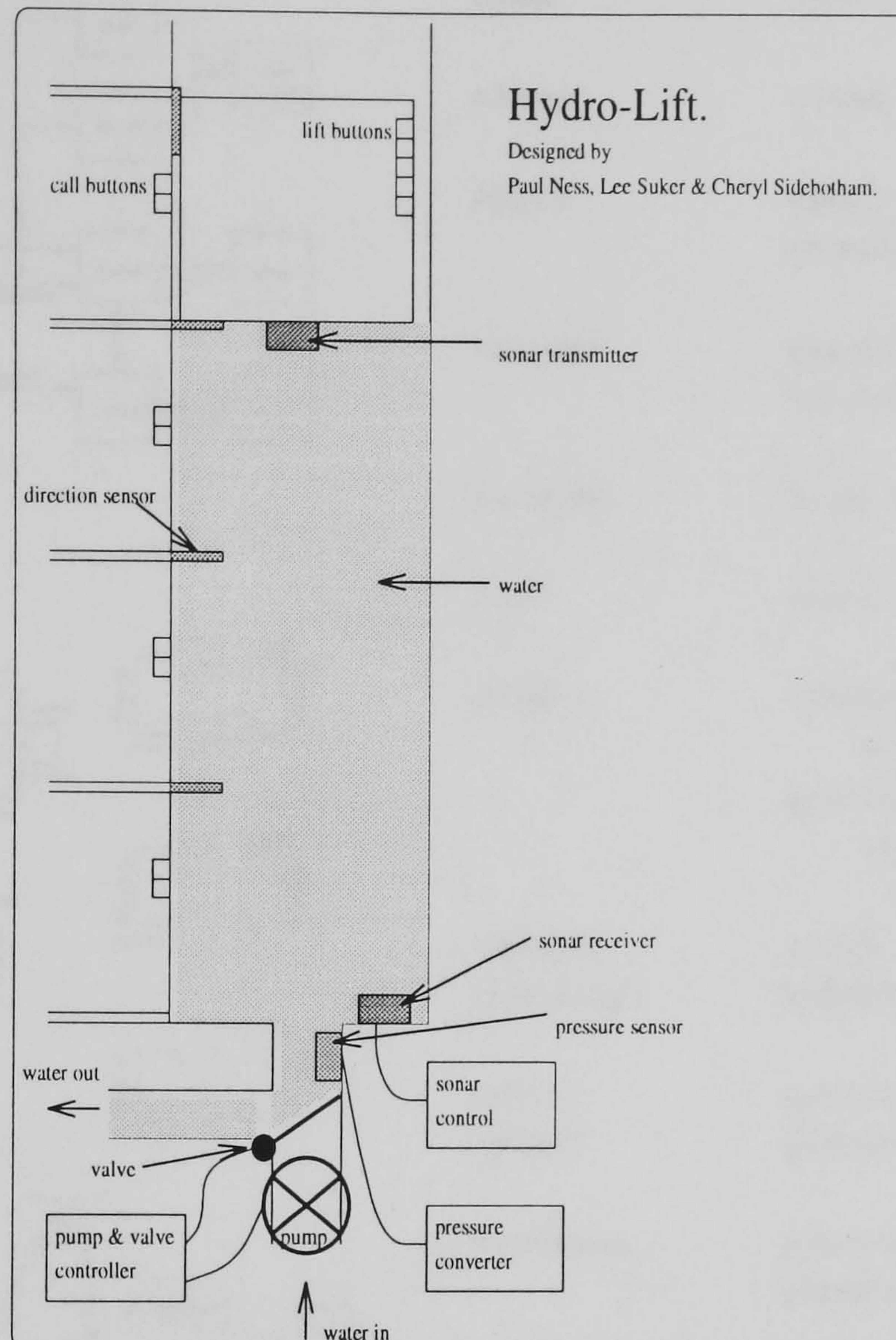
(d) Collects user 1.

The entities generated the redefinitions instead of the modeller.

The LSD specification was found to represent the subject in an intuitive way without any detailed representation of structure or function. The LSD specification was introduced in Chapter 2 as a statement consisting of agent definitions in which are specified the observables, derivates and protocols associated with the agent. It was argued in the previous chapter that these elements and their arrangement in the LSD specification reflect the observables and agents perceived by the modeller within the subject rather than the structure and function of the subject. The arrangement of elements in the LSD specifications of the lift project was found to correspond more to perceived agents and observables than to the structure of the subject or statements detailing the structure and function of the subject.



**Example 3.4. Sketches in PD.** The Hydrolift was first sketched out by designers in the lift project on paper and then using a general purpose line drawing application called *xfig*. The original computer-sketch

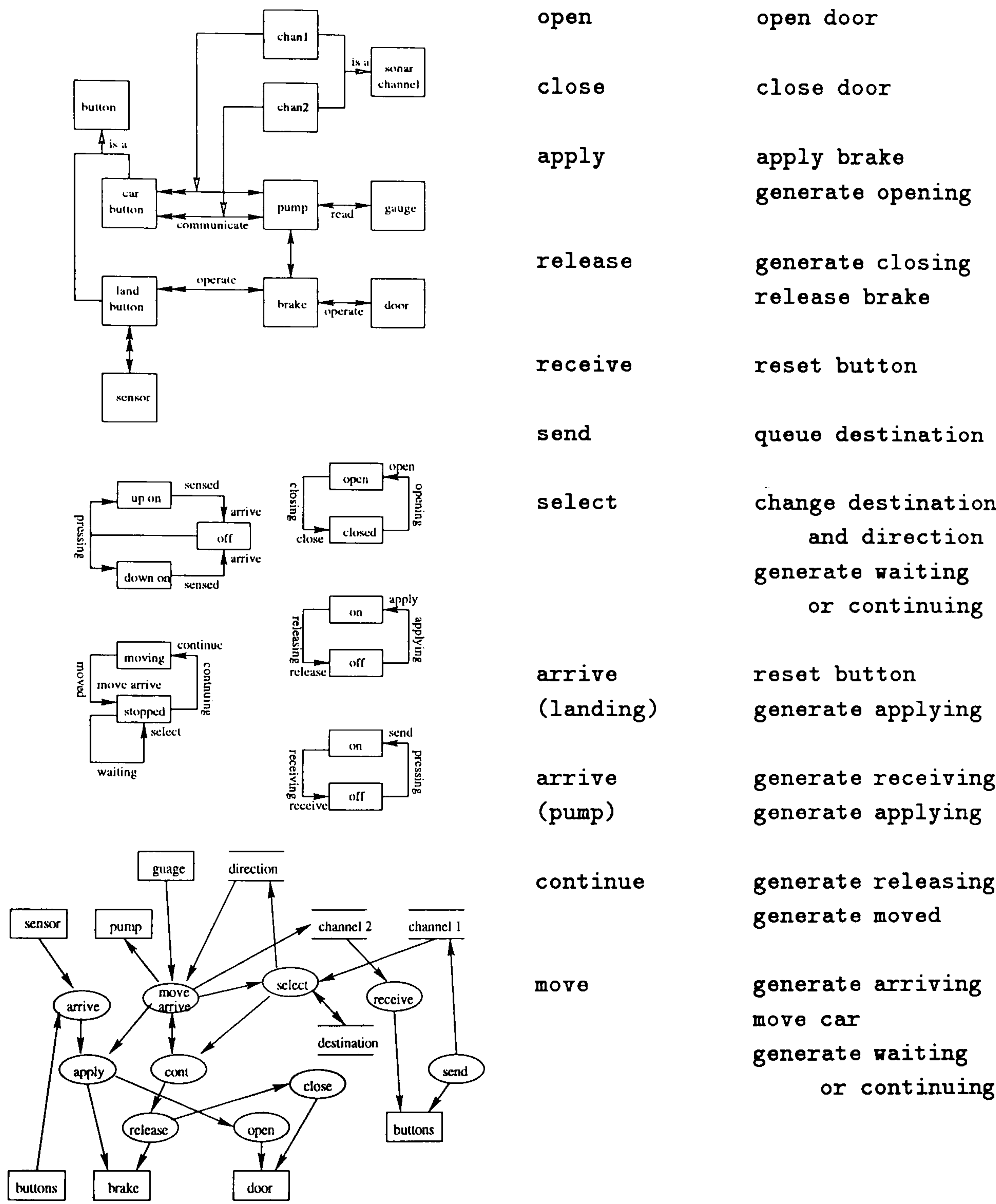


shows a realistic representation of the Hydrolift, albeit in caricature. The labels were not used in later sketches. Designers had to know the conventions for representing components, such as the crossed-circle representing the Hydrolift pump.

The visualization, sketch and structure model represented the structure of the subject in the lift project. Each element in the visualization and sketch was a caricature of the corresponding element in the subject. The modellers and designers used conventions for representing observables and components rather like a “graphical language” in the sense of Ferguson [Fer92] (Section 4.4.2). The structure model in SD serves as a graphical language to represent the structure of the subject, but this representation is more abstract than that in the EM visualization and PD sketch, where the actual arrangement of elements is reflected pictorially.



**Example 3.5. Structure, behaviour and process models in SD.** The structure, behaviour and process models and associated action definitions, shown here as pseudo-code



were the principal artefacts used by software developers in the lift project. The statement of requirements, another artefact used by software developers, is discussed later in this chapter.

The visualization and behaviour models in the lift project were found to represent the states of the subject. The visualization represents the states of the subject as values of DoNaLD variables corresponding to observables in the LSD specification. The visualization was found to improve on the single-state sketch by allowing the modeller to explore the states of the subject by changing the values

of DoNaLD variables. The behaviour model represents the states of the subject as symbols in the behaviour model. It was found that the correspondence between the state symbols and the states in the subject was less significant in SD than in EM. The role of state symbols in the behaviour model was found to be as place-holders for abstract state transitions, whereas the visualization and animation represent the states of the subject directly using a visual metaphor.

It was discovered in the lift project that the animation and process model represented the function of the subject. The animation represents the function of the subject as ADM entities corresponding to instances of agents in the LSD specification. The entities redefine DoNaLD variables thus animating the visualization. The behaviour of ADM entities is prescribed by the ADM script and the implementation of the `tkeden` interpreter. In principle, the behaviour of the ADM entities could be defined in a behaviour and process model. The actions in the ADM script are conditional state changes corresponding to the dataflows of the process model that map onto the state transitions in the behaviour model.

### 3.3 Subjects

A link between the suitability of artefacts for representing the subject and the nature of the subjects was discovered in the lift project. The term *subject* is used in this thesis to mean the system being modeled, designed or analyzed. Whether the subject was novel or familiar to the modeller, designer or software developer was found to significantly influence their activities. For a given novel or familiar subject, some artefacts were found easier to construct, and thus considered more suitable for representing the subject, than others.

The subject of the lift project was lift systems. The systems modelled, designed and analyzed were the SUL, MUL and Hydrolift:

- the SUL is a conventional lift system, from the viewpoint of an individual using the lift (the artefacts based on this personal viewpoint are termed in SUL artefacts);
- the MUL is a conventional lift system, from the combined viewpoints of many



individuals using a lift (the artefacts based on this multiple viewpoint are termed MUL artefacts);

- the Hydrolift is an innovative concept for a lift system, from the imagined viewpoint of an engineering designer (the artefacts based on this engineering viewpoint are termed Hydrolift artefacts).

Essentially, the SUL and MUL were familiar whereas the Hydrolift was, at least to begin with, novel to the modellers, designers and software developers in the lift project. For a detailed description of the lift project subjects see the SUL, MUL and Hydrolift statements of requirements in Appendix C.

It was discovered in the lift project that the suitability of artefacts for representing the subject depended on whether the subject was novel or familiar. The artefacts of EM, PD and SD can be ordered based on this dependency, starting with the artefact found most suitable for representing novel subjects and ending with the artefact found most suitable for representing familiar subjects:

1. LSD specification
2. visualization and sketch
3. animation
4. structure model, behaviour and process model

The LSD specification was found to be most suitable for representing novel subjects, such as the Hydrolift at the start of modelling. The SD artefacts were found to be most suitable for representing familiar subjects, such as the SUL, MUL.

This link between the suitability of artefacts for representing the subject and the novelty or familiarity of the subject provides insight into the use of artefacts in the mental process of conceptualization. It was argued in the previous chapter that EM reflects the process of conceptualization [Bey97]:

1. interaction with artefacts: identification of persistent features and contexts;
2. practical knowledge: correlation between artefacts, acquisition of skills;

3. identification of dependencies and postulation of independent agency;
4. identification of generic patterns of interaction and stimulus-response mechanisms;
5. non-verbal communication through interaction with similar environment;
6. situated use of language;
7. identification of common experience and objective knowledge;
8. symbolic representation and formal languages: public conventions for interpretation.

By mapping the stages of conceptualization onto the earlier sequence of artefacts we see that the PD and SD artefacts address different stages of conceptualization: the sketch is most suited to representation mid-way through conceptualization and the SD models are most suited to representation towards the end of conceptualization. Perhaps most significant is that the LSD specification and visualization appear better suited than the other artefacts to representing the subject during the earliest stages of conceptualization.

### 3.4 Actions

It was discovered in the lift project that each kind of artefact had a standard repertoire of actions associated with it. The activities of EM, PD and SD can be thought of as sequences of situated actions performed by modellers, designers and software developers as they strive to fulfill their goal of representing the subject. Their actions are the means of attaining their goal given the constraints posed by their environment [Sim81]. An association between artefacts and actions emerged by observing different modellers, designers and software developers constructing artefacts in different environments in the lift project.

The actions of constructing the LSD specification in the lift project were found to reinforce the modellers' beliefs about the subject. The modellers added agent definitions and refined existing oracle, derivate and protocol definitions in



the LSD specification, as shown in Example 3.6. The nature of the refinements were found to be, not so much corrections to what was represented, as shifts in the level of detail at which particular elements of the subject were represented, reflecting a better understanding of the subject by the modeller. Conviction about the validity of the model seemed to stem from the direct correspondence between the subject as perceived by the modeller and the representation of the subject in terms of observables and agents in the LSD specification.

The visualization, animation and sketch were typically constructed in the lift project by modellers and designers performing actions in an exploratory fashion. This was typically a two phase operation with modellers and designers first generating the artefacts and then evaluating the artefact against the subject, as shown in Examples 3.6 and 3.7. The generative phase of EM and PD in the lift project typically involved actions informed by the previous evaluation of the artefact:

- actions to change the number, position and appearance of shapes in the visualization by redefining the values of DoNaLD variables on-the-fly;
- actions to change the number and behaviour of ADM entities in the animation by redefining the values of ADM variables on-the-fly;
- actions to change the number, position and appearance of shapes representing components in the sketch by redrawing parts.

It was discovered that the actions of the modellers and designers were typically directed at refining the details of the artefacts whilst keeping the main structural and functional framework intact. This lent integrity to the artefact throughout the changes by maintaining the correspondence between artefact and subject.

**Example 3.6. Generative and exploratory actions in EM.** The Hydrolift was constructed by modellers in the lift project based on the MUL artefacts. The MUL oracles and handles of the car and shaft agents

`floor direction brake destination`

are associated with the mechanics of a lift system whereas the Hydrolift oracles and handles of the pump, sonar and sensor agents

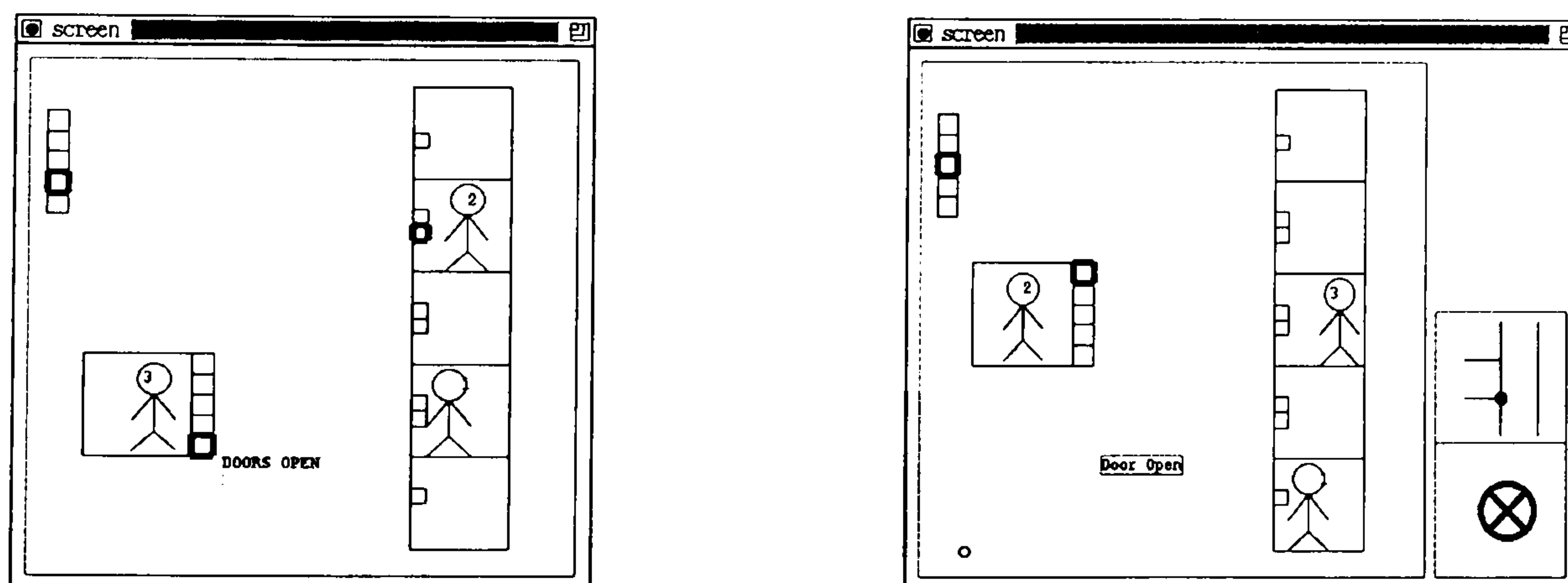
`pressure chan1 chan2 direction sensed`

are associated with hydraulics and communication through fluids. The need to link the car, shaft, pump, sonar and sensor in the Hydrolift resulted in the creative exploration of alternative interpretations of the MUL observables:

- the floor could be interpreted as the pressure of the column of liquid at the base of the shaft;
- the direction could be interpreted as the signal from a direction sensor;
- the destination could be interpreted as a target pressure.

From this process of generating and exploring interpretations emerged the LSD specification for the Hydrolift.

The Hydrolift visualization (shown right) was constructed in the lift project by modellers adding DoNaLD definitions, representing shapes corresponding to a pump and valve, to the MUL visualization (shown left).



The Hydrolift animation was constructed by modellers adding ADM entities, corresponding to the pump and valve, to the Hydrolift visualization.

Having generated the new artefact the modellers and designers would evaluate it as a representation of the subject. Both modellers and designers explored the correspondence between artefact and subject through imaginary interactions with the subject. In parallel with this the designer performed thought-experiments [Kap64] based on the sketch. The modeller, on the other hand, was able to perform



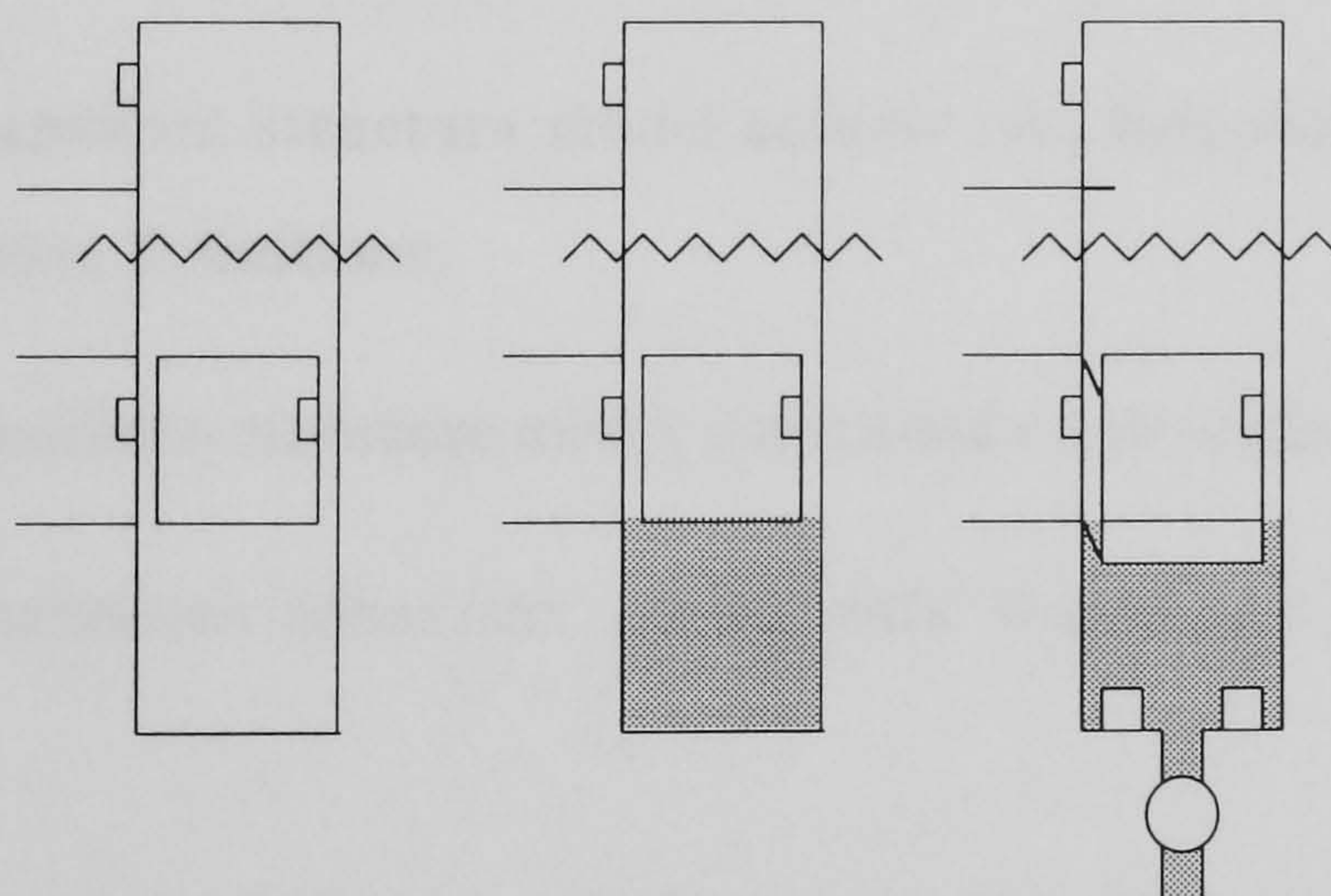
physical experiments using the visualization and animation:

- actions performed in parallel on the visualization/animation and subject to test correspondence;
- actions to setup similar scenarios in the visualization/animation and the subject with a view to testing the correspondence between artefact and subject for exceptional behaviours;
- actions to search for inconsistencies between the visualization/animation and the subject.

Such exploration typically suggested further changes to the artefacts to improve the representation of the subject. Inadequacies in the artefacts were found to diminish as modelling and design progressed with more time being spent by modellers and designers on refining details of the representation than on exploration.

---

**Example 3.7. Generative and exploratory actions in PD.** The following three sketches



show the three steps in designing the Hydrolift:

1. the designer retrieved their sketch for the MUL;
2. the designer added a representation of water filling the shaft;
3. the designer explored the composition and added more detail to the sketch in the form of appropriate components, such as a pump and sonar.

This approach to sketching the Hydrolift was found to keep the basic structure of the MUL.

---



It was found that software developers in the lift project constructed the structure, behaviour and process models by performing actions that were transformational in nature, as shown in Example 3.8. Exploratory type actions were generally found more likely to lead to ambiguities and inconsistencies in the models, so tended to be avoided, if at all possible, by the software developers in the lift project. The construction of models involved actions that transformed symbols:

1. actions to transform nouns into Object classes;
2. actions to transform noun phrases into structural relations between Object classes;
3. actions to transform verbs into actions;
4. actions to transform verb phrases into functional relations between Object classes;
5. actions to transform structure model Object classes into behaviour model state machines;
6. actions to transform structure model actions into behaviour model action labels and process definitions;
7. actions to transform structure model functional relations into state transitions;
8. actions to transform behaviour model state transitions into process model dataflows;
9. actions to transform behaviour model actions into process model actions;
10. actions to transform structure model Object classes and attributes into process model message sources and sinks and data stores.

It can be seen from the above list of transformations performed by software developers that the relative position of nouns and verbs within noun and verb phrases was important. These were typically written within a statement of requirements for the subject.



---

**Example 3.8. Transformational actions in SD.** It was discovered in the lift project that software developers generally constructed structure, behaviour and process models by transforming parts of other models or a statement describing the subject. For example, the requirements for the lift

... The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user ... The shaft mechanism releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

was used by software developers as the basis for transformations to construct the structure, behaviour and process model of the brake:

- The nouns “brake” and “shaft” were transformed into object Class representations in the structure model.
- The verb phrase “the door is opened and closed by the brake” was transformed into a functional association between the brake and door Object classes in the structure model.
- The verb phrase “The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied” was transformed into a functional association between the brake and shaft Object classes in the structure model.
- The phrase “The brake is applied whenever the car arrives at a landing requested by a user” was transformed into a functional association between the brake and lift button Object classes in the structure model.
- The verbs “applied” and “released” were transformed into the actions and transitions of the brake Object class represented in the behaviour model.

It was discovered that verb phases, functional associations, state transitions and dataflows had essentially the same meaning in SD. In effect, the software developer was expressing the meaning of model elements in different languages by transforming them.

---

### 3.5 Constraints

It was discovered that the timing and choice of actions during the process of EM. PD and EM was largely determined by the constraints imposed by elements within the environment. The constraints are the conditions for goal attainment [Sim81]. Pugh identifies a number of boundaries in total design that constrain the activities of designers [Pug91, Pug96]:

- the *business design boundary* represents the constraints placed upon the supply of technology and techniques to the design process by the elements of the business structure which include management, planning, organization and control;
- the *personal design boundary* (or interpersonal design boundary) represents the constraints placed upon the design process by the personal characteristics and skills of the designers which include their abilities to question existing practices, involve themselves in new disciplines and to communicate;
- the *product design boundary* is represented by the product design specification (PDS) that defines in detail the wide variety of constraints shown in Table 3.1 to be placed upon the design of a product.

Similar technical and non-technical constraints were identified in the lift project. The activities of modellers, designers and software were largely determined by what information and techniques they had available to them as well as those involved and the specifications of the subject.

Computer technology was found to be more of a constraint on the actions of modellers during the lift project than on the actions of designers and software developers. The designers and software developers used a simple line drawing application to construct their artefacts in the lift project. The **tkeden** interpreter, underlying the visualization and animation, is a computer-based tool that has the potential to support far more sophisticated modes of interaction. But visualizations and animations have to be simple given the current computer and **tkeden** interpreter technology. Although alternative technologies are being considered it seems that this limitation will always exist if the desired flexibility of the EM tools is to be maintained.



environment	ergonomics	scientific disciplines
company constraints	development costs	timescale
quality and reliability	maintenance	competition
product life span	life in service	materials
quantity	aesthetics	performance
standards and specification	profitability	research and development
customer	packaging	shipping
size	weight	market constraints
manufacture	product cost	safety

Table 3.1: Elements of the product design specification

The constraints on techniques was found to be most severe upon the actions of the software developer in the lift project. Pugh refers to the techniques available to a designer as their “tool-kit”, that includes techniques of analysis, synthesis, decision making, costing and modelling. The tool-kit of the software developer in the lift project was essentially limited to the standard object-oriented method of analysis:

1. transform the statement of requirements into a structure model;
2. transform the structure model into a behaviour model;
3. transform the behaviour model into a process model.

This method orders the transformational actions of the software developer: constructing the structure model uses actions 1 to 4; constructing the behaviour model uses actions 5 to 7; constructing the process model uses actions 9 to 10. The method presumes the existence of an unambiguous and consistent statement of requirements describing the subject. When such a statement exists the method of analysis can be a very powerful tool to the software developer.

The actions of the modellers and software developers were found to be constrained by specifications of the subject in the lift project, as shown in Examples 3.10 and 3.9. Pugh describes the PDS as a having a number of essential qualities:

- the PDS is a comprehensive and unambiguous specification of the product to be designed;

- the PDS constrains the actions of developers both subliminally during the generation of a sketch and consciously during the evaluation of a sketch;
- the PDS is an evolutionary document which, upon completion of the design activity, has itself evolved to match the characteristics of the final product;
- the PDS is written as “short, sharp definitive statements” rather than in “essay” form.

The LSD specification is the same as the PDS in all these respects except that the LSD specification is used by modellers in order to constrain the construction of visualizations and animations. The statement of requirements is similar to the PDS only it represents a more severe constraint on the actions of the software developer: the statement of requirements should be complete (complete is stronger than comprehensive) and not change during the construction of models.

---

**Example 3.9. Statement of requirements as constraint in SD.** When used, the statement of requirements was found to constrain the software developer. The pattern of nouns (bold) and verbs (*italic*) and associated nouns and verb phrases were found to determine the transformations performed by the software developer in the lift project.

On each **landing** there is an up and a down **button**. In the **car** there is a **button** for each **floor**. **Users** *make requests* for the **car** to *come* to their **landing** or *go* to another **landing** by *pressing* these **buttons**. The shaft **mechanism** *moves* the **car** towards a destination **landing** *stopping* whenever the **brake** is *applied*. The **brake** is *applied* whenever the **car** *arrives* at a **landing** *requested* by a **user** (for **requests** from **landings** the **direction** matters). On *arriving* at the destination **landing** the shaft **mechanism** *selects* the next **destination**. The shaft **mechanism** *releases* the **brake** and *starts* the **car** *moving* again. For safety the **door** is *opened* and *closed* by the **brake** ensuring that the **door** is only **open** whilst the **brake** is **on**.

The statement of requirements constrained the software developer to which transformations were performed. The sequence of these actions was constrained by the software development method.

---



---

**Example 3.10. LSD specification as constraint in EM.** It was found in the lift project that the LSD agent definitions, such as the definition of the shaft

```

agent shaft() {
state
  floor destination direction
oracle
  brake
handle
  brake
derivate
  direction is (floor < destination) ? UP :
               (floor > destination) ? DOWN : NIL
protocol
  brake == OFF -> floor = floor + direction,
  brake == ON && direction != NIL -> brake = OFF
}

```

seemed to be an absolute and unambiguous representation of the subject with few alternative representations emerging during the project. There seemed to be a direct and obvious correspondence between the agents and observables in the LSD definitions and those perceived within the subject:

- the shaft agent maps onto the notion of the mechanism responsible for raising and lowering the car;
- the observables map onto what the modeller thinks the shaft must be sensitive to or be able to act upon in order to move the car.

This authoritative definition of the subject was found to limit the activities of the modellers constructing the visualization and animation to represent the observables and agents within the LSD specification.

---

## 3.6 Environments

The constraints on the actions were found to be determined by the elements within the environment of the modellers, designers and software developers in the lift project. Constraints result from the presence of physical entities:

- the scientific and engineering knowledge recorded in papers, books, reports and such like determine the technological constraints;
- the people and tools available determine the constraints on techniques;
- the people in a group determine the interpersonal constraints;

- the specification document determines the constraint on the development of the system.

It was found in the lift project that EM, PD and SD were each characterized by the objects and people present within their respective environments. In particular, the environments were found to consist of a large number artefacts either archived or in the process of construction.

The emphasis of the environment of the modeller was found to be on the subject of lift systems. The environment of the modeller in the lift project consisted mainly of material linked with the current subject as, for example, the Hydrolift:

- books, papers, articles and project reports on lift systems;
- design sketches of the Hydrolift;
- computer running visualization or animation of the Hydrolift;
- LSD specification of the Hydrolift;
- computer archives of visualizations and animations of the SUL and MUL;
- archives of LSD specifications of the SUL and MUL;
- discussions between modellers about lift systems.

In addition, archives of information about EM was available in case the modeller had problems with one of the notations, tools or methods of representation. However, the environment of the modeller during the lift project consisted mainly of elements linked with the subject of lift systems.

As in EM, the emphasis of the designer's environment was found to be on the subject of lift systems. Ferguson describes a context for design in 1899, as shown in Example 3.11, that is similar in character to the design environment in the lift project:

- material for immediate reference including papers, sheaves and rolls of blueprints, and printed material;
- original full-size drawings stored in draws for immediate retrieval;



- data books and sketch books;
- consultations between designers;
- photographs and actual finished designs.

A modern design environment is typically computer-based. This has a number of advantages that are associated with the computer being able to store and manipulate components representations. This facility was not used in the lift project because of the simplicity of the designs and the novelty of the Hydrolift. An account of the negative impact of computers on innovation in product design, discussed at length by Ferguson and Pugh in [Pug91, Pug96, Fer92, Fer77], is outside the scope of this thesis.

The emphasis of the software developer's environment was found to be on the subject of SD. The environment of the software developer in the lift project consisted predominantly of material linked with SD. Similarities between the environment of the IBM WSDL (Appendix B) and the context for SD in the lift project were observed:

- books describing the object-oriented analysis and design method;
- printouts and screen displays of structure, behaviour and process models;
- discussions about improving the effectiveness of the method;
- computer tools for supporting the method;
- archives of software components for use in the design phase;
- detailed knowledge about the subject in the form of a statement of requirements or software for a system with the similar behaviour to the subject.

In the lift project, essentially all the information needed about the subject was in the statement of requirements. Similar characteristics were observed in the IBM WSDL project working environment.

**Example 3.11. Traditional environments in design.** Ferguson gives the Baltimore and Ohio Railroad drafting room in 1899 as an example of an archetypal product design environment before the computer workstation took over:

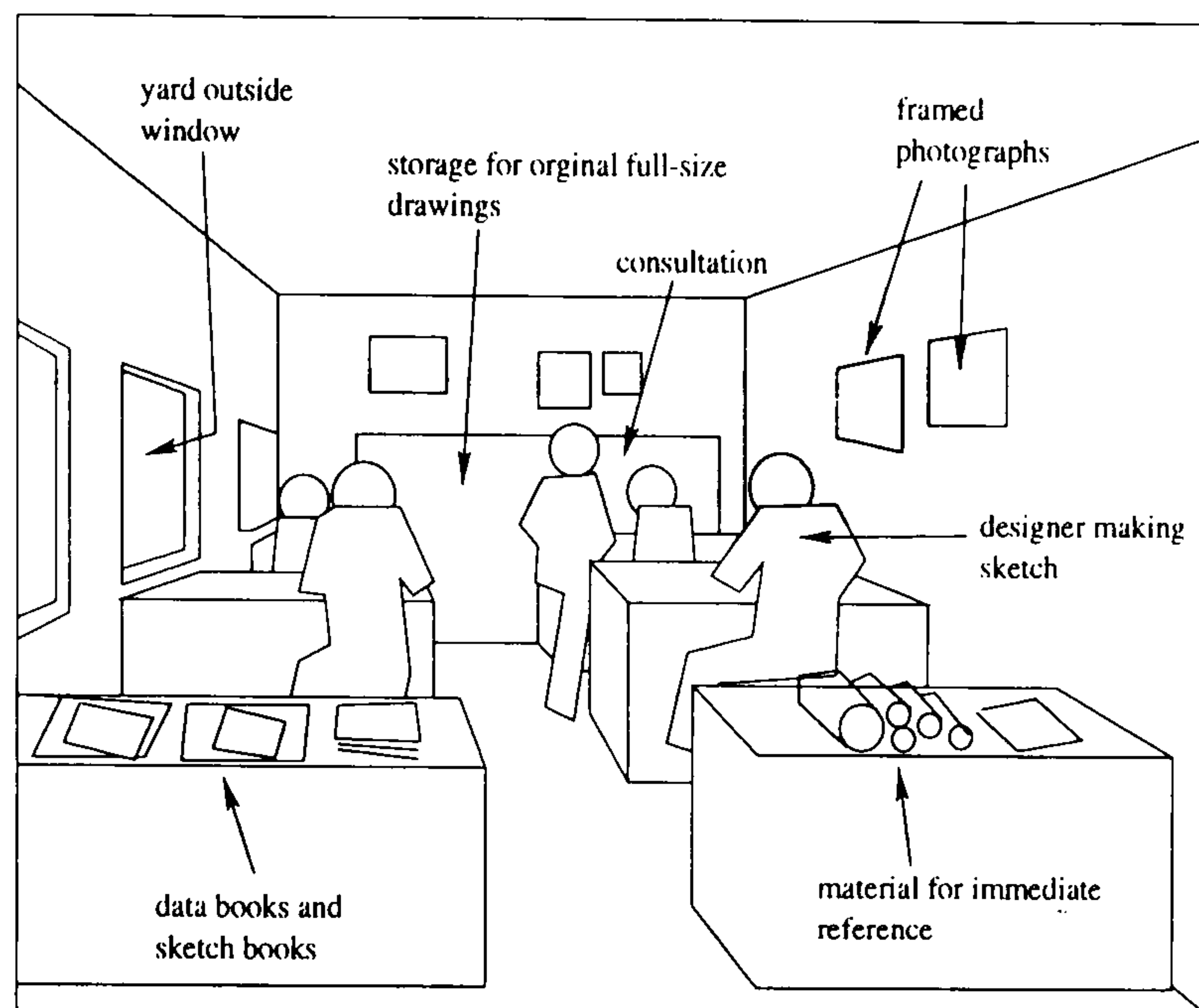


Figure 3.1: Baltimore and Ohio Railroad drafting room 1899.

“The photograph [copied in Figure 3.1] clearly shows where and how locomotives, rolling stock, and railroad structures were designed. The designer in the right foreground appears to be making a sketch that will be converted by a draftsman into a working drawing. He has for immediate reference papers, sheaves and rolls of blueprints, and printed materials. At the far end of the room, original full-size drawings are stored flat in drawers for immediate retrieval. Data books and sketch books are in the foreground. A consultation between two colleagues is taking place at the centre of the picture. The results of the designers’ and the draftsmen’s work are displayed in framed photographs on the walls, and one can walk into the yards to see the real thing. The designers are thus intimately in touch with the world they have designed, and they are engaged intellectually and physically at a detailed level in planning the future of their railroad” [Fer92].

### 3.7 Knowledge

It seemed that the modellers, designers and software developers used different kinds of knowledge in the construction of artefacts to represent the subject. The kind of knowledge appeared to reflect the nature of the artefacts:

- the modeller used knowledge about observables and agents to construct the LSD specification;



- the modeller and designer used knowledge about the structure, disposition and relationship between agents to construct sketches and visualizations;
- the modeller and software developer used knowledge about behaviour and function to construct the animation.

The knowledge used by modeller, designers and developers came from long-term memory and the immediate sensations of interacting with the artefacts and subject. It is this knowledge that informed the actions of the modellers, designer and developers in the lift project.

An appropriate way to view knowledge in the lift project is in terms of mental models. It is generally accepted among cognitive scientists that knowledge consists of mental models that mimic the characteristics of external entities [JL83, JL88, GS83]. Johnson-Laird provides a taxonomy of mental models [JL83]:

1. a simple relational model consisting of tokens and relations;
2. a spatial model is a relational model consisting of spatial relations;
3. a temporal model is a sequence of spatial models related temporally;
4. a kinematic model is a temporal model without temporal discontinuities;
5. a dynamic model is a kinematic model in which there are causal relations between certain spatial models contained within it;
6. a conceptual model is a dynamic models in which there are semantic and recursive relations.

The first five are termed physical models “in that, with the possible exception of causality, they correspond directly to the physical world. They can represent perceptible situations” [JL83]. The final mental model encompasses all conceptual models and is characterized by recursive relations and semantic relations that support the use of language.

There is a link between the characteristics of artefacts and mental models. It is possible to associate the artefacts of EM, PD and SD with the mental models described in the Johnson-Laird taxonomy:

- LSD protocols and agents correspond to dynamic and simple conceptual models respectively;
- visualizations correspond to dynamic models;
- animations correspond to kinetic models;
- components in a sketch typically correspond to dynamic models;
- structure, behaviour and process models correspond to conceptual models about structure and function.

It is expected that an agent corresponds to a dynamic model in which there are relations grouping causal relations. The SD artefacts would be expected to correspond to conceptual models that are complex systems of relations representing structure and function.

The notions underlying LSD protocols and agents were found to be familiar to everybody in the lift project even though the notions were difficult to define precisely. Perhaps this familiarity was because the general concepts underlying protocols and agents are reinforced whenever we perceive the consequence of an action or interact with another person. Causality and personification are widely recognized as being perhaps the most familiar of all concepts [HT95]:

- causality is the intuitive understanding that some regularities in the world are based on the relationship between causes and effects;
- personification means to treat something that is not a person as if it were one.

The concepts of causality and personification straddle the boundary between physical and mental models in Johnson-Laird's taxonomy. They are both difficult to visualize and difficult to put into words. However, they are recognized as playing an important role in the mental process of analogical transfer [FWS92, HT95].

The LSD specification has an important role to play in the conceptualization of novel subjects by encouraging the "mental leap" [HT95] from a physical to a conceptual model. Although the author can only surmise about the activity of the brain based on the evidence, it is plausible that the modeller has three kinds of mental models in his head when constructing and using an LSD specification:



- a physical model corresponding to the novel subject;
- a physical/conceptual model corresponding to protocols and agent definitions in an LSD specification;
- conceptual models in long-term memory.

This suggests that the LSD specification acts as a “conductor” for conceptual knowledge to flow to the novel subject so that it can start to be understood in terms of familiar concepts. This mental process is known as analogical transfer in which a relationship or set of relationships in one context is transferred to another, resulting in mental models that are analogous to those already familiar [FWS92]. In effect, the subject is created in the mind of the modeller through analogical transfer. Analogical transfer is an important part of common sense thinking: “[common sense] reflects an enormous amount of information that one has gained about the world and provides a large number of practical rules - many of them quite logical - for dealing with day-to-day life. It is so much part of everyday life that one seldom thinks about it” [Wol92]. Analogical transfer is also widely regarded as one of the most important processes in creativity [FWS92, Pug91, HT95].

### 3.8 Summary and conclusion

In this chapter, the disciplines of EM, PD and SD were characterized in terms of the artefacts, subjects, actions, constraints, environments and knowledge of each discipline. It was found that EM and the conceptual design stage of PD were similar in the lift project:

- the artefacts corresponded to the subject;
- the subjects were novel or familiar;
- the actions were generative and exploratory in nature;
- the constraints acted to limit rather than prescribe the activity;
- the environment emphasized the subject;

- the knowledge used was essentially “physical” mental models [JL83].

In the lift project, SD was found to be different from both EM and PD:

- the artefacts corresponded to abstract conceptual models of the subject;
- the subjects were typically familiar;
- the actions were transformational in nature;
- the constraints acted to limit and prescribe the activity;
- the environment emphasized the process of developing software;
- the knowledge used was essentially abstract conceptual mental models [JL83].

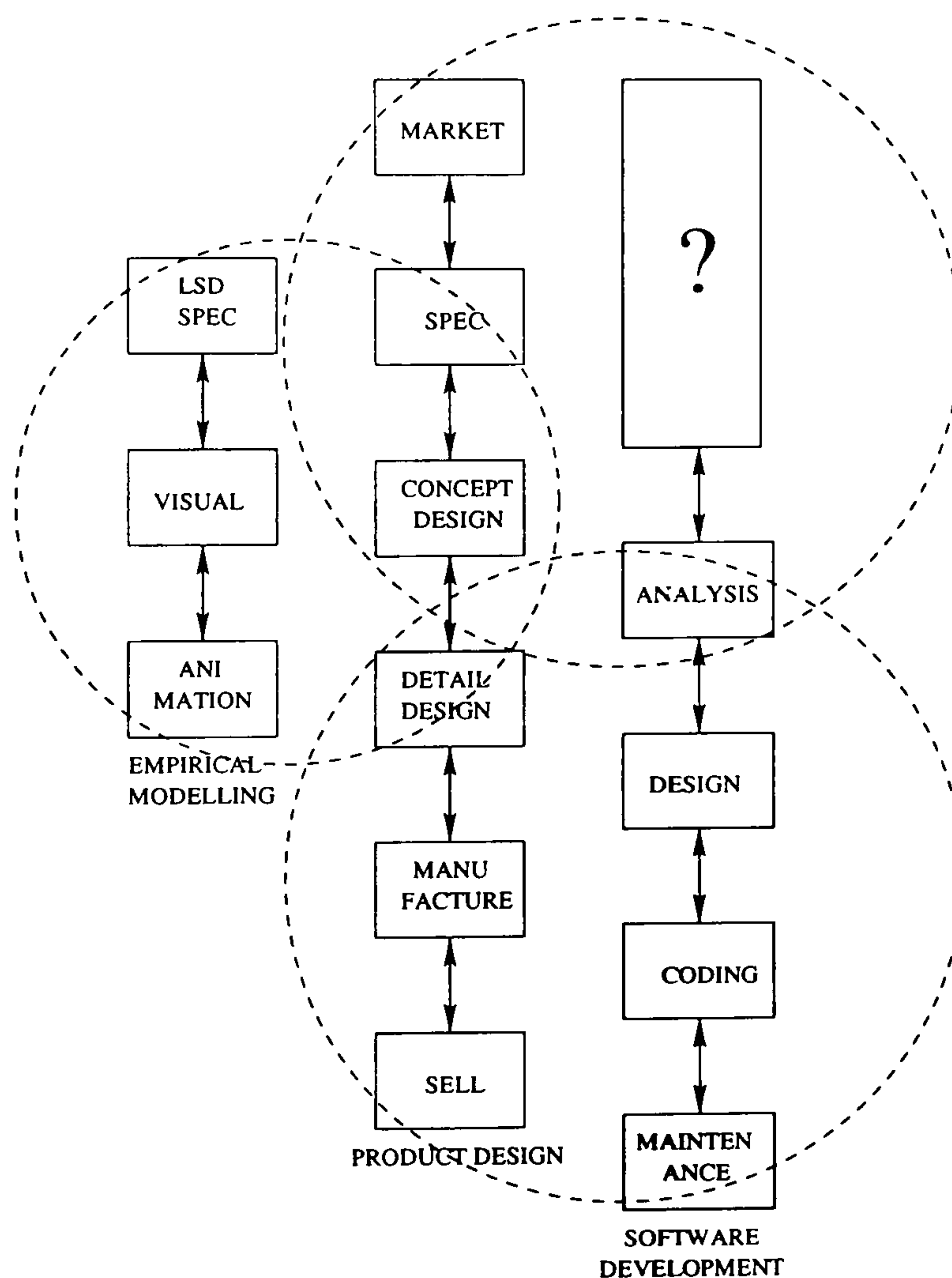


Figure 3.2: Comparison of EM, PD and SD.

These results are represented in Figure 3.2. It shows the similarity between EM and conceptual design identified here and in [ABCY94c, Car94, Bey89]. In addition, the figure shows the similarity between SD and the subsequent phases of PD, from detail design through to selling the product in the marketplace. The



construction of the models in SD parallels engineering in detail design, characterized by the use of analysis, prototyping and computer simulations [Pug91, Fer92].

Figure 3.2 shows an activity preceding SD that has the same characteristics as EM and conceptual design in PD. The results of this chapter suggest that artefacts that support creativity will be essential determinates of this question-mark activity. With this emphasis on creativity the new approach to software development may appropriately be termed “creative software development”.

## Chapter 4

# Artefacts of EM, PD and SD

Chapter 3 highlighted the importance of artefacts in determining the nature of EM, PD and SD. Further investigation of the nature of these artefacts requires an understanding of what makes them essentially different. This investigation demands a framework for identifying and contrasting the properties of artefacts.

This chapter compares the artefacts of EM, PD and SD to identify how they are essentially different. A framework is provided by a set of creative properties, characterized in the theory of creative cognition [FWS92], and their complementary analytical counterparts. The results of examining the artefacts of the lift project with respect to each of the properties are given. The characterization of artefacts is extended to construals for representing novel phenomena [Goo90] and engineering drawings [Fer92].

### 4.1 Definition

Established definitions of the term artefact provide an appropriate place to start defining what is meant by the term artefact as used in this thesis. The Webster dictionary [web13] provides two entries for the word artefact (or artifact):

1. a usually simple object (as a tool or ornament) showing human workmanship or modification.
2. a product of artificial character due to extraneous (as human) agency.



It is clear from these definitions that an artefact is something made by humans as opposed to naturally occurring. These definitions are of limited use because (1) is too specific and (2) is too general. A definition is needed that gives a better characterization of what an artefact is. Simon provides just such a definition by characterizing an artefact in terms of form (inner-environment), context (outer-environment) and purpose (the terms form and context are used by Alexander in a similar characterization [Ale67]):

**A.** an artifact can be thought of as a meeting point - an “interface” in today’s terms - between an “inner” environment, the substance and organization of the artifact itself, and an “outer” environment, the surroundings in which it operates. If the inner environment is appropriate to the outer environment, or vice versa, the artifact will serve its intended purpose [Sim81].

According to (A) an artefact is characterized, not so much by a physical object, but by an abstract boundary that divides the world into form and context. For example, the essential qualities of a cup, its purpose, to hold liquid, and its shape, approximately cylindrical, form an abstract characterization of the artefact. Since the abstract interface cannot exist in the world, it must be represented in the mind or by an artefact constructed for the purpose of representing knowledge. Norman [Nor91] defines a cognitive artefact:

**B.** an artificial device designed to maintain, display, or operate upon information in order to serve a representational function.

The artefacts defined in (B), which include books, drawings and computers, are commonly used to empower the cognitive processes of the human brain during the construction of artefacts. However, contemporary research of cognitive artefacts has typically explored their use in what is essentially non-creative contexts [FWS92]. Research on cognitive artefacts has tended to focus on how they support the mental processes of analysis, rather than creativity, when they are used to represent familiar artefacts. Since cognitive artefacts have mostly been studied with respect to the

construction of familiar products, such as road systems [Per95], they have arguably become associated with analysis.

In this thesis, the word artefact is used to mean the artefacts constructed during EM [BC95], PD and SD in the lift project:

- the LSD specification, visualization and animation of EM;
- the sketches of PD;
- the structure, behaviour and process models of SD.

Although the established meanings of the term artefact apply to these artefacts the definition of cognitive artefacts is perhaps the closest. However, because of its association with analysis, the term cognitive artefact perhaps best defines the artefacts of SD in the lift project. Since the artefacts of EM and PD were found to support creative cognition [FWS92] the artefacts in this thesis are simply referred to as “artefacts” without qualifying them as cognitive.

## 4.2 Need for artefacts and how they help cognition

Artefacts, such as those used during EM, PD and SD in the lift project, are needed to extend the limited information storage and processing capability of the human brain so that people, such as modellers, designers and software developers, can be more creative and analytical. The limitations of the human brain are well known in this respect:

- In the information processing model of the human brain memory is divided into short-term and long-term stores [HF75]. All new information is processed by short-term store before it becomes embedded in long-term store. The capacity and duration of short-term store is severely limited:
  - the duration of visual short-term store limited to a fraction of a second, even though the capacity is practically unlimited [Gre70, Gre94];
  - the capacity of verbal short-term memory limited to approximately seven units of information [Mil56] and its duration limited to approximately twenty seconds [HF75].



These limitations constrain the processing capability of short-term store and therefore reduce the flow of information into long-term store.

- Cognitive scientists [JL83, JL88, GS83] believe that people make predictions about the world by forming mental models: internal structures that represent the external reality in at least approximate ways [HT95]. Norman [Nor91] observes that mental models have their limitations:
  - mental models are incomplete;
  - people’s abilities to “run” their models is severely limited;
  - mental models have a limited duration before parts of them are forgotten;
  - similar mental models become confused;
  - mental models are kept simple so as not to exceed the limited capacity.

These limitations place constraints on what can be apprehended.

The artefacts of the lift project mirrored the short-term stores and mental models in the heads of the modellers, designers and software developers. In this way, the artefacts empowered the mental processes of analysis and creativity. Whether an artefact improved on analysis or creativity appeared to depend on the nature of the artefact.

### 4.3 Characterization of artefacts

In this section, the EM, PD and SD artefacts of the lift project are characterized in terms of the creative and analytical properties listed in Table 4.1.

The following definitions of creative properties are based on those given in [FWS92] (a review of this book is given in Appendix D) that describes them as being some of the most important for supporting creative cognition:

- **Novelty** is probably the most important. Although a familiar structure might be interpreted in creative ways the possibilities for creative discovery should be much greater if the structure is relatively uncommon to start with.

Creative	Analytical
Novelty	Familiarity
Ambiguity	Unambiguity
Implicit meaningfulness	Explicit meaning
Emergence	Completeness
Incongruity	Consistency and Congruity
Divergence	Convergence

Table 4.1: Creative and analytical properties

- **Ambiguity** should afford greater opportunities for creative exploration and interpretation. For this reason one might want to avoid imposing narrow interpretations onto the structures when they are being formed.
- **Implicit meaningfulness** is a general perceived sense of “meaning” in the structure. This sense of meaning is related to interpretation. Artefacts often seem to have a hidden underlying meaning to them which encourages further exploration and search.
- **Emergence** refers to the extent to which unexpected features and relations appear in the structure. These features and relations are not anticipated in advance and become apparent only after the structure is completely formed.
- **Incongruity** refers to the conflict or contrast among elements in a structure. This often encourages further exploration to uncover deeper meanings and relations in order to reconcile the conflict and reduce the psychological tension it creates.
- **Divergence** is related to ambiguity but refers more specifically to the capacity for finding multiple uses or meanings in the same structure. Something could be relatively unambiguous in terms of its underlying structure but still afford a variety of interpretations: “A hammer, for example, is a relatively unambiguous form but can be used in a variety of different ways - as a tool, a paperweight, a weapon and so on” [FWS92].



The following definitions of analytical properties were devised by the author as counterparts for the creative properties defined above. The properties of unambiguity, consistency and completeness are familiar in logic and mathematics to do with formal specifications [Win90, Sai96, Hal90, BH95, San88], convergence is based on the notion of convergent thinking [FWS92], explicit meaning is based on semantics, and familiarity is intended to be thought of as the opposite of novelty:

- In this thesis **familiarity** is the analytical counterpart of novelty. Familiarity is probably the most important analytical property because it implies there exists knowledge about the structure on which to base analysis.
- In this thesis **unambiguity** is the analytical counterpart of ambiguity. In the formal sense, a structure has the property of unambiguity if and only if it has only one meaning [Win90]. In this thesis, a structure is unambiguous when it provides little opportunity for creative exploration of its meaning.
- In this thesis **explicit meaning** is the analytical counterpart of implicit meaningfulness. In this thesis a structure has an explicit meaning when there is a general agreement about the meaning of the structure. This agreed meaning is typically represented as a statement in a commonly understood language, such as a natural language description or a C++ program. In such cases the meaning depends less on individuals and particular situations and more on symbols and conventions for representation.
- In this thesis **completeness** is the analytical counterpart of emergence. In the formal sense, a structure has the property of completeness if and only if inconsistencies in the structure can be detected by methods that are defined independently of the notion of truth [Hod77]. Most methods of analysis are defined independently of the notion of truth whereas creative exploration deals with truth. Formally, a structure is complete when all inconsistencies can be detected by analysis. Less formally, a structure is complete when searches fail to reveal any emergent features.
- In this thesis **consistency** and **congruity** are the analytical counterparts of incongruity. In the formal sense, a structure has the property of consistency



if and only if it is not possible to derive any contradictions from it [Win90]. In this thesis consistency is treated as the formal case of congruity. Congruity refers more generally to the sense of harmony among elements in a structure.

- In this thesis **convergence** is the analytical counterpart of divergence. In this thesis convergence is a property of a structure that promotes convergent thinking: “In *convergent thinking*, one goes from an initial problem state through a series of prescribed operations in order to converge upon a single correct solution. Convergent thinking is ideal for well-defined problems for which there is only one allowable conclusion” [FWS92]. In this thesis convergence is associated with a methodical process arriving at one of possibly a number of satisfactory solutions.

What follows is a characterization of EM, PD and SD artefacts of the lift project in terms of the creative and analytical properties defined above.

For convenience, the illustrative examples that go with this chapter have been organized into an appendix at the end of this chapter.

#### 4.3.1 Novelty and familiarity

The SUL and MUL were found easier to represent than the Hydrolift. The SUL and MUL were familiar to the modellers, designer and software developers in the lift project. The modellers were able to represent the structure and function of the SUL and MUL in visualizations and animations without first constructing an LSD specification. The designers were able to produce detailed sketches of the SUL and MUL components. The structure and function of the SUL and MUL were described in requirements statements, as shown in Example 4.1.

Little use was made of an LSD specification in constructing the SUL and MUL visualizations and animations. The SUL and MUL were modelled in the lift project by writing DoNaLD and ADM scripts directly. The modellers were already familiar with the way the lift system looked and functioned from their viewpoints as users so were able to turn these ideas directly into visualizations and animations, as shown in Example 4.2. The SUL and MUL LSD specifications were written after the construction of visualizations and animations had already commenced.



The approach taken by modellers changed when representing the Hydrolift, with more emphasis being placed on the construction of the LSD specification. The modellers used the LSD specification during the early stages of modelling to represent their first tentative interpretations of the Hydrolift, as shown in Example 4.3. The LSD specification was used by modellers to share their early interpretations of the subject with one another. It was subsequently used as the basis for constructing a visualization and animation.

Stating the requirements for the novel Hydrolift was found to be more difficult than stating the requirements for the familiar SUL and MUL. The problem was describing the subject in sufficient detail, for the software developer to construct structure, behaviour and process models, with no other reference than the vague idea of what a Hydrolift was. The requirements for the Hydrolift were stated by describing detailed EM and PD artefacts that represented the structure and function of the subject in terms of geometrical shapes and prescribed behaviours.

### 4.3.2 Ambiguity and unambiguity

There always seemed to be multiple interpretations of the subject when it came to stating the requirements in the lift project, as shown in Example 4.4. This suggests that the SUL, MUL and Hydrolift were ambiguous with respect to natural language. It was found difficult to decide between alternative statements indicating that there was no clear correspondence between the elements in the statement of requirements and the elements in the subject it described. It is this lack of correspondence between the domain of natural language and the domain of lift systems that made the statement of requirements in the lift project ambiguous.

Although essentially ambiguous with respect to general interpretation, the statement of requirements was found to be unambiguous with respect to the specific interpretation of structure and function. Analysis of a statement of requirements resulted in few alternative structure, behaviour and process models, as shown in Example 4.5. This was probably due to the direct correspondence between the elements in the statement of requirements and the elements in the SD models. For example, nouns and verbs in the statement of requirements corresponded to classes

and actions in the structure model.

Few ways were found of describing the SUL, MUL and Hydrolift in an LSD specification suggesting that the subjects of the lift project were essentially unambiguous with respect to LSD, as shown in Example 4.6. Few significant changes were made during the construction of visualizations and animations indicating a direct correspondence between the LSD specification, in terms of observables and agents, and the subject as perceived by the modeller. It is this correspondence between the domain of LSD and the domain of lift systems that made the LSD specifications in the lift project unambiguous.

It was found that the artefacts of EM and PD afforded reinterpretation similar to that of the subject. The artefacts preserved the essential ambiguity of the subject through mimicry. This allowed the artefacts created by modellers and designers at one stage of the lift project to be creatively reinterpreted later in the lift project in place of the subject, as shown in Example 4.7. For example, the artefacts constructed to represent the MUL were reinterpreted in the construction of the artefacts to represent the Hydrolift.

### 4.3.3 Implicit meaningfulness and explicit meaning

The meaning of the structure, behaviour and process models was found to be essentially independent of the subject. The meaning of each model was defined explicitly in terms of the other models and the statement of requirements, as shown in Example 4.8:

- the meaning of the process model was defined in terms of the statement of requirements and the behavioural model;
- the meaning of the behavioural model was defined in terms of the statement of requirements and the structure model;
- the structural model was defined explicitly in terms of the statement of requirements.

The dependency between the statement of requirements and the subject was not passed on to the models through transformation. The nouns and verbs, that give the



statement of requirements its meaning, were treated as symbols in the construction of the structure, behaviour and process models: the verbs and nouns from the statement of requirements were listed then the positioning of the verbs and nouns in the statement of requirements was analyzed.

The analysis of the statement of requirements was found to be simple in comparison to formulating it in the first place. Stating the requirements meant repeated interpretation of the SUL, MUL and Hydrolift until a consistent description of the structure and function of the subject was achieved. EM and PD artefacts were found to help in this process of interpreting the subject in terms of nouns, verbs, phrases and sentences. It was found easier to construct and describe EM and PD artefacts, with the same sense of meaningfulness as the subject, than it was to write a statement of requirements based on the subject alone.

The EM and PD artefacts were found to have the same sense of implicit meaningfulness as the subject they represented, as shown in Example 4.9. This meaningfulness was given by the direct correspondence between artefact and subject. The obvious similarities between sketches, visualizations and animations meant that their meaningfulness to modellers and designers in the lift project was not surprising. The meaningfulness of the LSD specification was more surprising because it was neither visual nor interactive like the subject. The direct correspondence between the LSD specification and the subject as perceived by the modeller seemed to give it a similar depth of meaning as the visualization and animation.

#### 4.3.4 Emergence and completeness

The EM artefacts in the lift project were found to have the property of emergence, as shown in Example 4.10. Modellers in the lift project would think they had finished an artefact only to find emergent features indicating incompleteness. Features emerged during exploration that were not intentionally included within the artefacts by the modellers. Construction of EM artefacts can be thought of in two phases:

1. representing the obvious features of the subject in the LSD specification, visualization and animation;



2. exploring the LSD specification, visualization and animation to discover emergent features and searching for the features in the subject.

The LSD specification was found to be least helpful in discovering emergent features because it represented elements of the subject that were obvious to the modellers including observables and agents. The visualizations and animations, however, represented structural and functional features of the subject that were not obvious at the start of modelling and only emerged in the subject after repeated generation and exploration of artefacts. Emergent features discovered by modellers in visualizations and animations were subsequently incorporated into the LSD specifications.

The statement of requirements was found to be complete in the sense of Hodges [Hod77] discussed above. It was complete with respect to the structure, behavioural and process models, as shown in Example 4.11. Essentially, the structure of the statement of requirements contained the information necessary to construct the structure, behaviour and process models without having to explore the meaning of its contents. All inconsistencies in the models could be found by examining the structure of the statement of requirements without having to consider the truth of the statement. This meant that the SD statement of requirements and structure, behaviour and process models were complete.

#### 4.3.5 Incongruity and congruity

The concept of the Hydrolift was chosen for its incongruity. Based on the familiar notion of a conventional lift system, the SUL and MUL contained few conflicting elements. The Hydrolift combined the incongruous elements of water and conventional lift systems.

The designer began by constructing an incongruous sketch of the Hydrolift then continued by resolving the conflicts between elements within the sketch, as shown in Example 4.12. The designer first sketched the MUL with water filling the shaft. The resulting artefact preserved the same sense of incongruity as the imagined subject by juxtaposing representations of a conventional lift system and water. The rest of the design of the Hydrolift involved adding components representations, including a pump and sonar, to resolve the conflicts.



A similar approach was taken by modellers in the lift project. Instead of adding a representation of water, the modellers added definitions of agents associated with water, including a pump and sonar, to the MUL LSD specification. The resulting specification preserved the same sense of incongruity as the imagined subject. The rest of the modelling involved the reinterpretation of MUL observables and reassigning of definitions and protocols to new pump and sonar agent definitions, as shown in Example 4.13.

The approach taken by modellers and designers in describing the Hydrolift could not be used in SD. The MUL statement of requirements and models could not be meaningfully juxtaposed with representations of water and Hydrolift components. An important property of the SD artefacts was consistency which is related to completeness discussed earlier and defined at the beginning of this chapter. The artefacts had to be descriptions of the Hydrolift after all the inconsistencies about structure and function had been resolved. This was achieved in the lift project by interpreting the congruous EM and PD Hydrolift artefacts once they had been constructed.

#### 4.3.6 Divergence and convergence

SD artefacts encouraged convergence towards the goal of an operational model of the subject. The SD process typically began with a statement of requirements. This was transformed into a structure model that was then transformed into the behavioural model that was finally transformed into the process model by following the SD method of analysis. Each transformation brought the software developer closer to their goal of an operational model of the subject. By following the method of analysis in the lift project the software developers were guaranteed to converge upon their goal of an operational model of the subject.

The artefacts of EM encouraged convergence towards as well as divergence from the goal of an animated model of the subject. The modeller generally took convergent steps whenever possible in the lift project by adding ADM entities to circumscribe the behaviour of visualizations. However, the modellers also explored artefacts to discover alternative behaviours in the LSD specifications and visualiza-

tions that were subsequently evaluated against the subject. By taking convergent and divergent steps the modeller was able to achieve his goal of an animation by side-stepping obstacles that he encountered on the way, such as limitations in the tools and insufficient knowledge about the subject.

Pugh recommends the method of “controlled convergence” for conceptual design based on the use of decision matrices described in Chapter 2: “[controlled convergence] allows alternate convergent (analytic) and divergent (synthetic) thinking to occur, since as the reasoning proceeds and a reduction in the number of concepts comes about for rational reasons, new concepts are generated. It is alternately a generative (creative) and a selection process.” [Pug91, Pug96]. This design activity parallels convergence and divergence in EM.

## 4.4 Further characterizations of artefacts

In this section the construal [Goo90] and design drawings, in the sense of Ferguson [Fer92, Fer77], are characterized in the same way as the artefacts of the lift project.

### 4.4.1 Construals

In [Goo90] Gooding explores how scientists share their experiences of a novel phenomenon in the absence of an existing framework for interpretation:

I argue that when negotiating agreement about what they are seeing (as distinct from personal experience) observers exchange tentative constructs or *construals* of their personal experience ... Construals are a means of interpreting unfamiliar experience and communicating one’s trial interpretations. Construals are practical, situational and often concrete. They belong to the pre-verbal context of ostensive practices ... my purpose is to draw attention to the neglect of something important in the history of science and probably to learning generally, namely, how observers bring unruly experience into the domain of public discourse ... This book is about how such experience moves from an observer’s private world into the domain of discourse and argument [Goo90].



So, the purpose of a construal is to provide a means of interpreting novel experience and communicating trial interpretations. Gooding [Goo90] describes a variety of construal types that were used by Faraday in his experiments to understand electromagnetic phenomena:

- Sketches that “conveyed, through an image, aspects of experience that had been (or was being) made sense of,” in particular, Faraday is famous for his representation of the magnetic field around conductors in the form of concentric circular directed arrows.
- Apparatus - a wooden dowel with an arrow drawn on it, wires and magnets - that was manipulated by Faraday in order to model the dynamic aspects of his experience.
- Words were used by Faraday to describe construals when the communication of experience over space and time was necessary, however, the description was not used in place of the construal only as an approximation to it.

Gooding [Goo90] also describes in some detail the typical context for construals. The context must consist of at least one observer for the construal to fulfill its purpose. In addition, the context consists of the means of generating construals and the influences that determine its eventual form: “Discovery takes place in a context replete with resources and motivations, images, models, assumptions, percepts, values, instruments, techniques, goals, allies, rivals, enemies, and so on” [Goo90].

The emphasis on purpose over form, and the variety of possible forms and contexts, suggests that a construal can be appropriately thought of as an artefact in the sense of Simon [Sim81]. With this in mind it should be possible to identify creative and analytical properties of construals. Certainly in [Goo90] Gooding provides some evidence for the emergence of creative properties:

- The argument that construals provide nonverbal “reference points” for language suggests the creative property of implicit meaningfulness, also that it becomes “easy to see” phenomena in terms of construals and they pave the way for the “self-evidence” of experience.

- Stating that “correspondences” between words and what they denote “emerges” as an instance of construing indicates the creative property of emergence.
- Saying construals “may be compatible with several theories or with none” indicates a variety of creative properties including inconsistency.

It is the presence of these properties that support creative exploration of the subject that determines the success of a construal: “The outcome of exploration with a construal will decide whether it is developed, held as an unexplored possibility, or abandoned ... the few that pass into the interpretative vocabulary of science do so in virtue of their heuristic, communicative and instrumental value.”

In addition, Gooding [Goo90] provides some evidence of the analytical properties that emerge in experience with respect to a highly developed construal:

- The following suggests the analytical property of explicit meaning emerging as the result of generating a construal:

Some construals survive and become interpretations whose reference is gradually stabilized in terms of established observational practices. As interpretations they engage theoretical assumptions and problems ... the agency that produced it disappears [Goo90].

- Arguing that “interpretations are more literary and more theory-oriented versions of construals” suggests that construals converge upon, yet do not quite become, unambiguous, consistent and complete representations of phenomena.

So, as the development of a construal progresses more and more analytical properties emerge in the experience of the phenomena by the observer.

#### 4.4.2 Design drawings

In [Fer92] Ferguson gives an account of how the creation of drawings help designers clarify their ideas for products and communicate their ideas to others:

In order to produce a new machine, structure, or other technological artefact, two separate but closely related processes are generally required.



In the first, engineering designers convert the visions in their minds to drawings and specifications. In so doing, they solve an ill-defined problem that has no single ‘right’ answer but has many better or worse solutions. Engineers learn a great deal during the process of design as they strive to clarify the visions in their minds and seek ways to bring indistinct elements into focus. When the designers think they understand the problem, they make tentative layouts and drawings, analyze their tentative designs for adequacy of performance, strength, and safety, and then complete a set of drawings and specifications. Those who will make or build the machine, structure or system can learn exactly what they are expected to produce. Until their task is complete and the project has been turned over to the user, those drawings and specifications will be the formal instructions that guide their work [Fer92].

So, drawings and specifications represent the final products of a conversion process from the idea in the mind of the designer to physical realizations of it. Ferguson [Fer92] identifies three forms of sketches that help the designer in this process:

- “The first is the *thinking sketch*. Leonardo’s [da Vinci] notebooks contain dozens of such sketches, and a host of later engineers have used sketches to focus and guide nonverbal thinking.”
- “The next is the *prescriptive sketch* which is sometimes scaled and which is made by an engineer to direct a drafter in making a finished drawing.”
- “The third kind of sketch, produced constantly in exchanges between technical people, is the *talking sketch*.”

Each type of sketch is used in a different context: the thinking sketch is private to a designer, the prescriptive sketch is public and the talking sketch is between a group of designers. The talking sketch is unusual because it is neither private nor public: “Such sketches make it easier to explain a technical point, because all parties in a discussion share a common graphical setting for the idea being debated” [Fer92].

It has already been previously argued in this chapter that design sketches are artefacts and that creative and analytical properties emerge in the design during the

creation of a sketch. There is evidence in [Fer92] to support this argument. There is evidence for creative properties:

- Saying that “some of the choices will have been wrong” suggests the creative property of incompleteness in the design process.
- “Making wrong choices is the same kind of game as making the right choices; there is often no *a priori* reason to do one thing rather than another, particularly when neither has been done before” suggests the creative properties of ambiguity and novelty.
- The statement that “various members of a design group can be expected to have divergent views of the most desirable ways to accomplish the design they are working on” indicates the creative property of divergence.
- “The precise outcome of the [design] process cannot be deduced from its initial goal” suggests the creative properties of ambiguity, inconsistency and incompleteness.

In addition, there is evidence in [Fer92] for analytical properties with respect to prescriptive sketches and finished drawings:

- “Engineering drawings are expressed in a graphic language, the grammar and syntax of which are learned through use; it also has idioms that only initiates will recognize” suggests the analytical property of explicit meaning in finished drawings.
- Saying that “because the drawings are neatly made and produced on large sheets of paper, they exude an air of great authority and definitive completeness” indicates the analytical property of completeness.
- Stating that drawings are “precise” suggests the analytical properties of unambiguity, consistency and completeness.

So, as the design process progresses more and more analytical properties emerge in the ideas of the designers for the new product.



## 4.5 Summary

In this chapter the artefacts of EM, PD and SD were compared. It was found that the SD artefacts in the lift project had the properties of

- familiarity,
- unambiguity,
- explicit meaningfulness,
- completeness,
- consistency, and
- convergence

The artefacts of EM and PD were found to have the properties of

- novelty,
- ambiguity,
- implicit meaningfulness,
- emergence,
- incongruity, and
- divergence

and yet still have some of the properties of the SD artefacts. This suggests that the artefacts of SD are essentially analytical in nature whereas the artefacts of EM and PD support creativity as well as analysis.

Finally, this chapter extends the characterization of artefacts to construals [Goo90] and design drawings, in the sense of Ferguson [Fer92, Fer77]. The characterization uncovers similarities between construals, sketches and EM artefacts.

## Appendix: Illustrative examples for Section 4.3

---

**Example 4.1. Familiarity in stating requirements.** The statement of requirements for the SUL

On each landing there is a button and in the car there is a button for each floor. The user makes a request for the car to come to his landing by pressing a button. The shaft mechanism moves the car to his landing and opens the door. The user enters the car and presses a button. The shaft mechanism moves the car to the landing he requested and opens the door. The user exits the car. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

shows a familiarity with the details of the individual components and how they function and interact with one another.

---



**Example 4.2. Familiarity in EM.** The SUL visualization shown in Appendix C and the ADM shaft entity

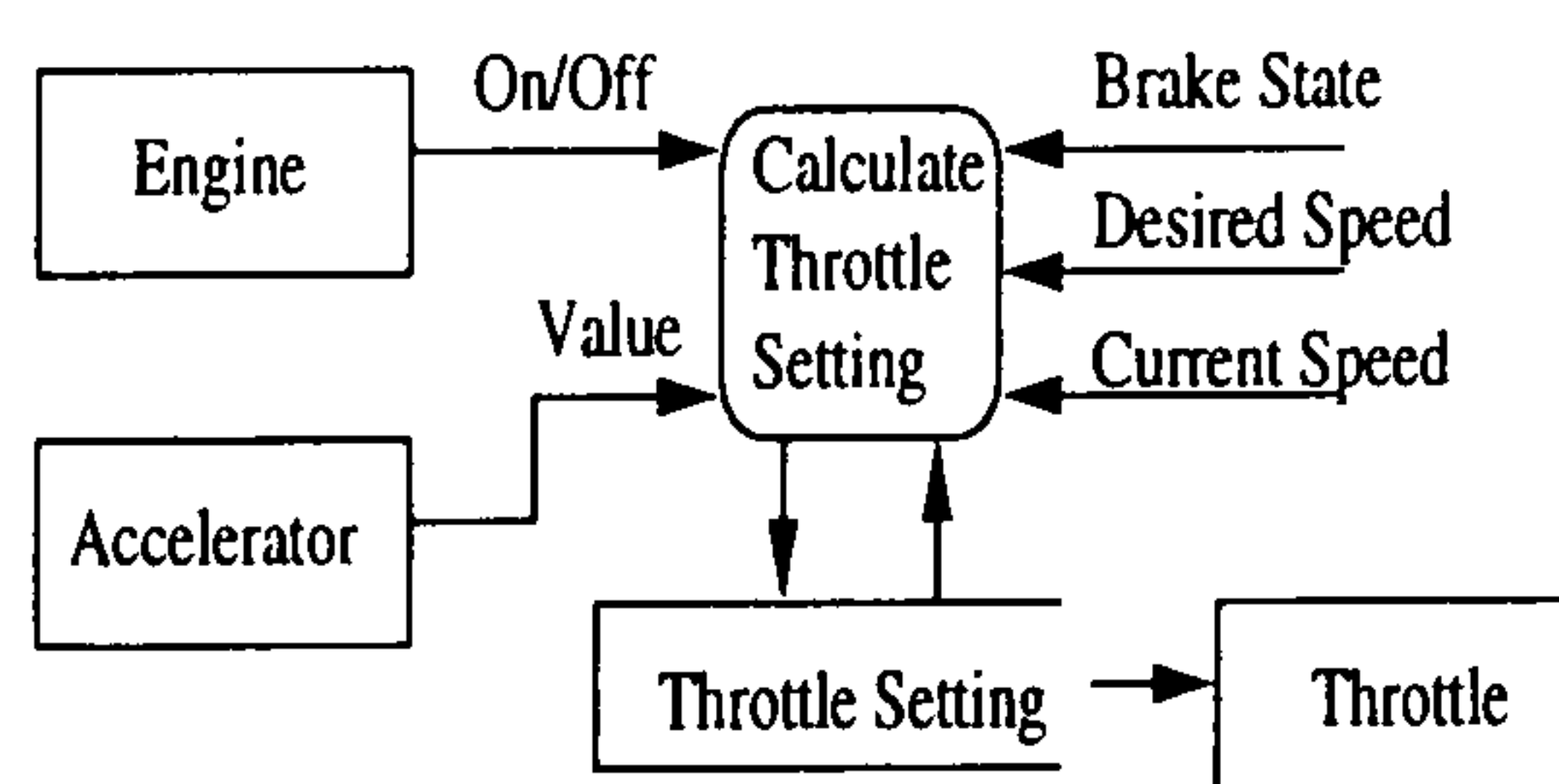
```
entity shaft() {
  definition
    direction is (floor < destination) ? UP :
                (floor > destination) ? DOWN : NIL
  action
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}
```

show a familiarity with the structure and function of the SUL:

- the shaft consists of five floors;
- if the break is on and the lift car has a destination then the next action is to release the break.

Both the visualization and ADM script represent decisions about the shape and function of the shaft that are not represented in the LSD description.

In the VCCS and digital watch projects [BBY92, BC95] the function and structure of the subjects were familiar to the modellers. The object-oriented definition of the VCCS [Boo86, Deu88, Deu89] and Statechart definition of the digital watch [Har88] showed a familiarity with both devices. The EM projects involved implementing the formal models directly in definitive languages without much use of LSD. There are clearly parallels between part of the data flow diagram for the VCCS [Boo86]



and the outline for the throttle manager agent

```
agent throttle_manager {
  state
    throttleStts throttlePos
  oracle
    measSpeed cruiseSpeed cruiseStts engineStts accelStts
  handle
    throttleStts
}
```

suggesting that the familiar functional detail of the VCCS given in [Boo86] was represented directly in LSD and subsequently in EDEN.

---

**Example 4.3. Novelty in EM.** The LSD specification of the SUL shaft agent, prior to the construction of the visualization and animation

```
agent shaft() {
  state
    floor destination direction
  oracle
    brake
  handle
    brake
  derivate
    direction is (floor < destination) ? UP :
                (floor > destination) ? DOWN : NIL
  protocol
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}
```

lacks the commitment to structure and function that comes from familiarity with the subject:

- no indication of the shape of the shaft;
- no detail of the behaviour of the shaft.

LSD was found suitable for representing aspects of the subjects that were not yet familiar to the modeller. Although the above definition appears similar to the subsequent ADM script the status and meaning of the LSD specification is very different from the ADM script as mentioned in Chapter 2.

LSD was made use of during the early stages of the classroom interaction project when the combined behaviour of pupils and teachers was unfamiliar to the modeller. It was fairly straightforward for the modeller to identify pupils as agents, but behavioural aspects were less well understood. The modeller continued by attributing states, oracles and handles to the pupil:

- the location of the pupil in the classroom (`location` state),
- the activity that the teacher is engaged in (`teacherActivity` oracle), and
- the ability for a pupil to change their mind (`memory` handle).

Descriptions of the behaviour of pupils and teachers emerged during EM as the modeller became more familiar with the behaviour of pupil and teacher interaction through the process of observation and experiment. This knowledge was subsequently represented in visualizations and animations of the classroom.

---



---

**Example 4.4. Ambiguity in stating requirements.** The subject of the MUL brake was found to be ambiguous with respect to stating its requirements with many ways of describing the brake emerging during the lift project. For example, statements describing the car stopping included the following

- “The shaft mechanism stops at a landing then the brake is applied”.
- “The shaft mechanism begins stopping before the destination landing, allowing the car to gently decelerate, then the brake is applied”.
- “The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied”.

Each statement means something slightly different but all describe the same observation of the car stopping. The last statement was the one actually used in the statement of requirements for the MUL.

---



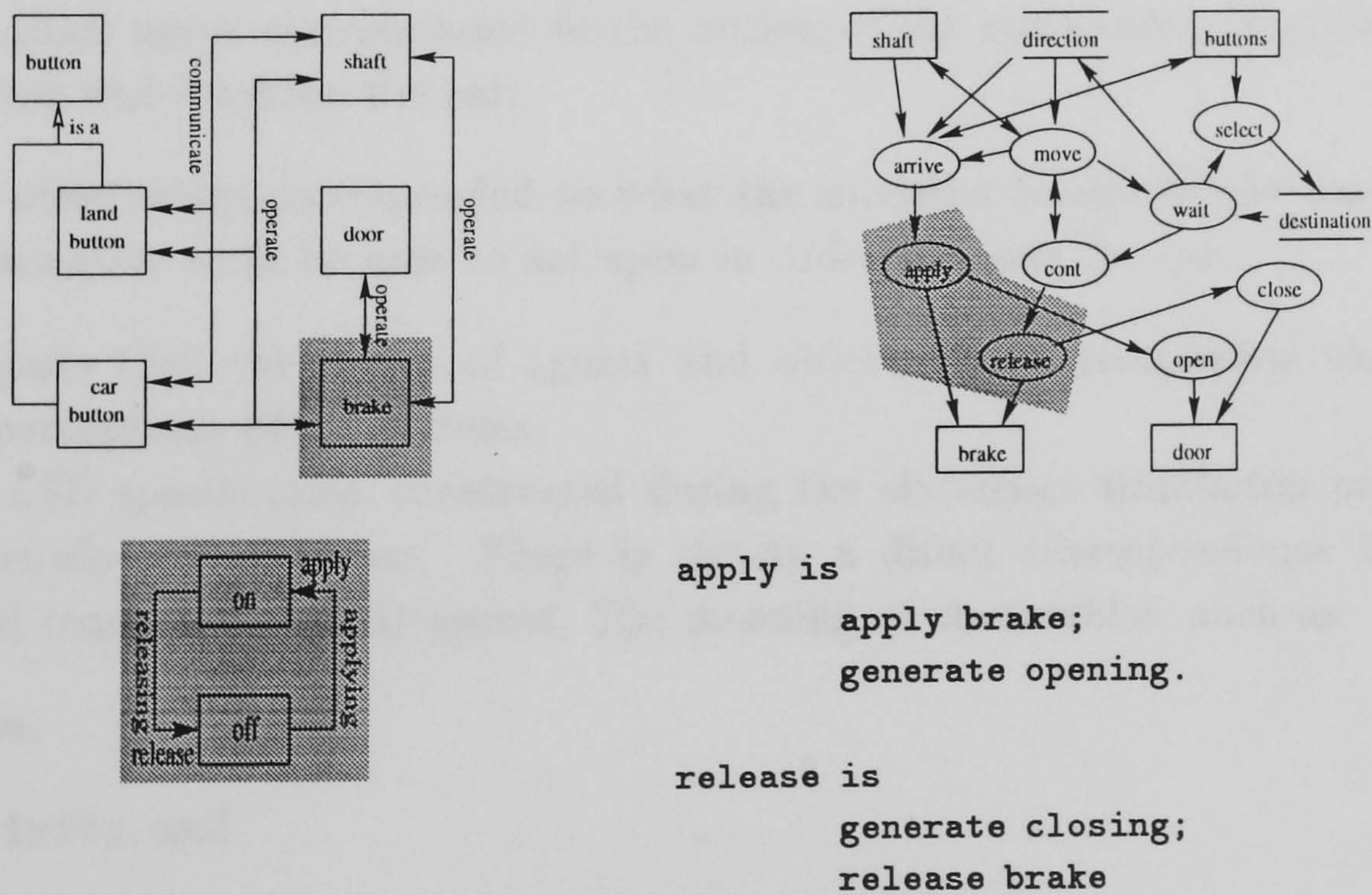
**Example 4.5. Unambiguity in SD.** The MUL requirement for the brake

... The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user ... The shaft mechanism releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

was found to be unambiguous with respect to the representation of structure, behaviour and function. By analyzing the structure of the statement, and largely ignoring the content, by treating the nouns and verbs as essentially meaningless tokens, it was found that

- the “brake” is related structurally to the “shaft”, “door” and “buttons”,
- the “brake” is “applied” by the “buttons” and “released” by the “shaft”, and
- the “brake opens” and “closes” the “door”.

This information was used to construct the model for the brake



that explicitly shows the structural and functional relations identified in the analysis of the requirements.



---

**Example 4.6. Unambiguity in EM.** It was found in the lift project that the LSD agent definitions, such as the definition of the shaft

```
agent shaft() {
  state
    floor destination direction
  oracle
    brake
  handle
    brake
  derivate
    direction is (floor < destination) ? UP :
                (floor > destination) ? DOWN : NIL
  protocol
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}
```

seemed to represent the subject unambiguously, with respect to the subject, with few alternative representations emerging during the project. There seemed to be a direct correspondence between the agents and observables in the LSD definitions and those perceived within the subject:

- the shaft agent corresponded to the notion of the mechanism responsible for raising and lowering the car;
- the observables corresponded to what the modeller imagined the shaft must be sensitive to or be able to act upon in order to move the car.

This suggests that the choice of agents and observables corresponded closely to peoples' perceptions of lift systems.

The LSD specification constructed during the classroom simulation project is also essentially unambiguous. There is clearly a direct correspondence between pupils and teachers and LSD agents. The meaning of observables, such as

- name,
- activity, and
- ability

have relatively unambiguous common-sense meanings. or teachers.

---

---

**Example 4.7. Ambiguity in EM.** The first two guards of the MUL LSD specification of the landing

```

agent landing(_F) {
  state
    landButton
  oracle
    floor direction brake
  handle
    brake destination
  protocol
    landButton[_F] == UP && _F == floor + 1 && brake == OFF -> brake = ON,
    landButton[_F] == DOWN && _F == floor - 1 && brake == OFF -> brake = ON,
    landButton[_F] != OFF && direction == NIL -> destination = _F,
    floor == _F -> landButton[_F] = OFF
}

```

represent the observation of the car arriving at the landing. At a conceptual level there is little ambiguity as to what this observation means. However, at a more detailed level this observation may be made in many different ways. There is scope for creative exploration of the alternatives. For example, in the specification of the same two guards in the Hydrolift landing

```

landButton[_F] == UP && sensed[_F - 1] == UP && brake == OFF
landButton[_F] == DOWN && sensed[_F + 1] == DOWN && brake == OFF

```

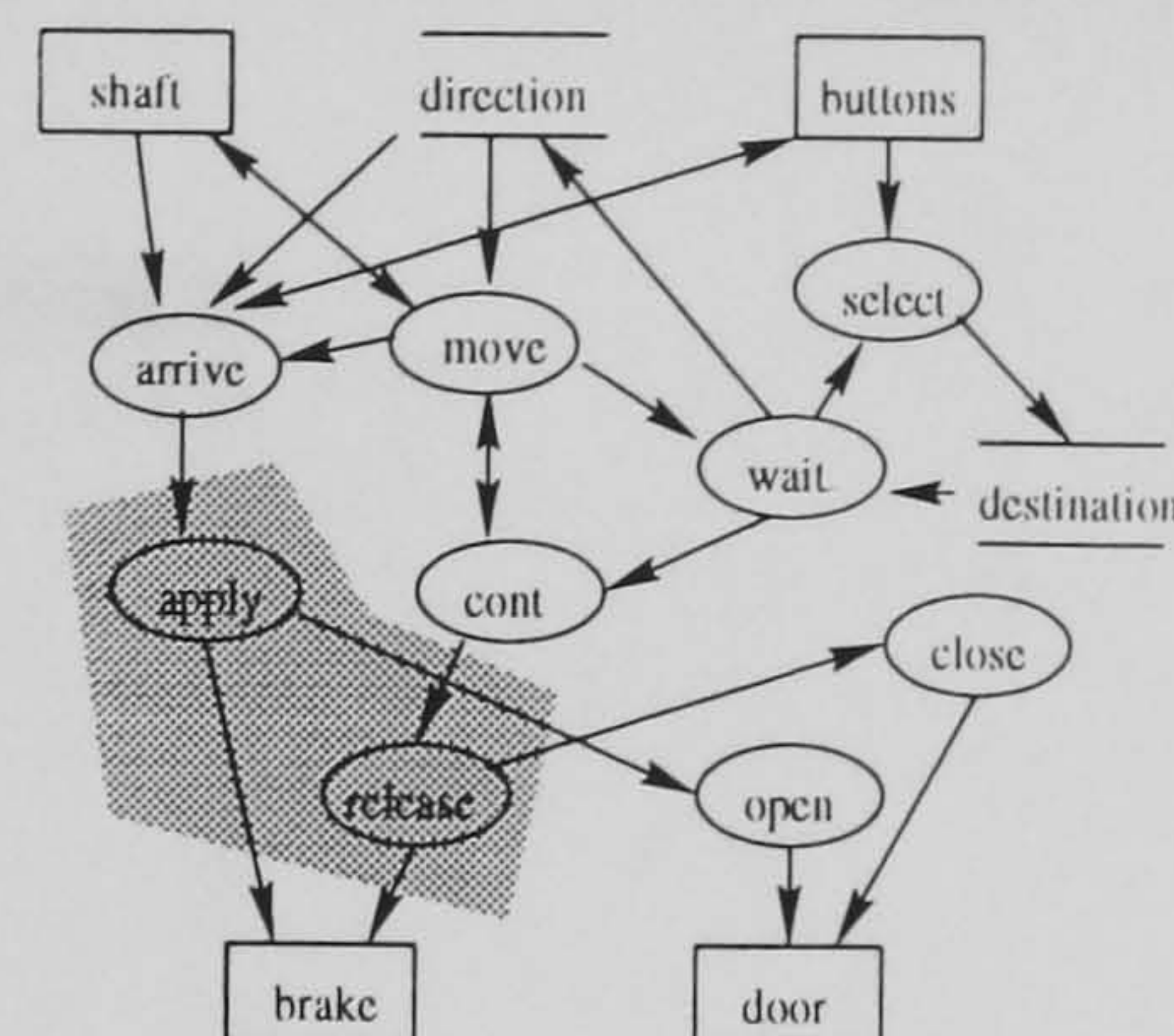
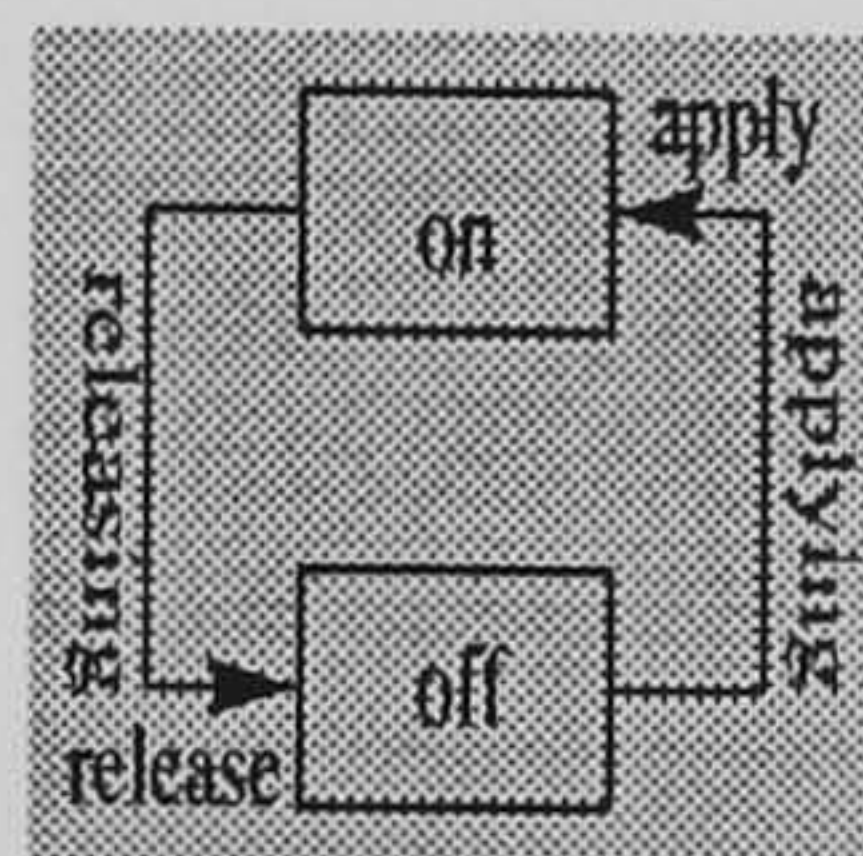
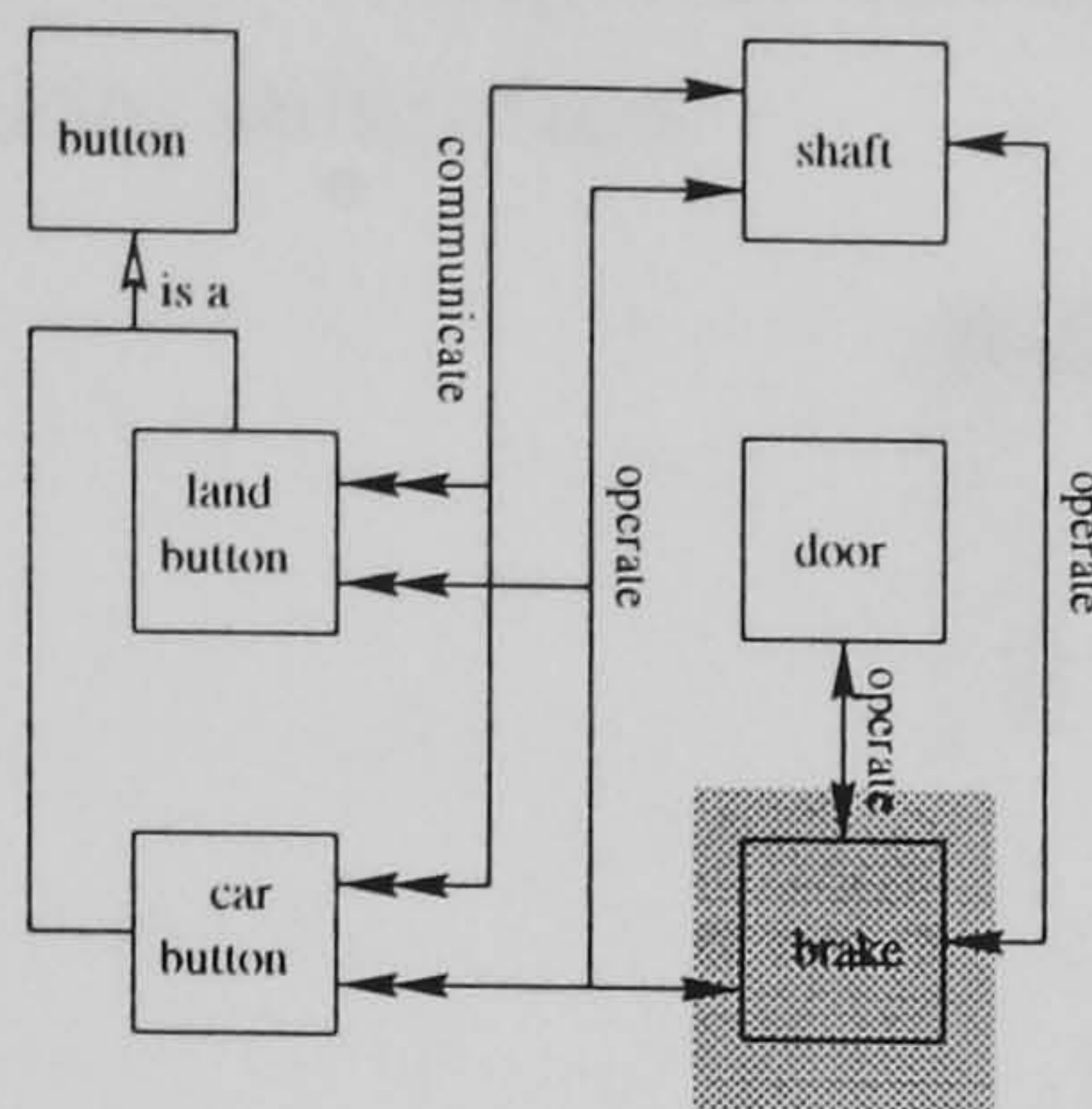
the observation is realized by a sensor that is attached to the side of the shaft and detects the direction of travel of the car.

Ambiguity in EM artefacts generally arise when the modeller cannot observe a feature of the subject. In the classroom simulation project the modeller is able to describe the observables of pupils and teachers quite unambiguously because they can be observed through experimentation. This was not so for the definition of the “decision function”: “the decision function is the part most crucial to modelling the realistic behaviour of pupils because it decides what the pupil will do next depending on the situation they are currently in and the contents of their memory” [Dav96]. The definition of the decision function is more ambiguous because it represents the workings of the mind that are inherently unobservable and difficult to conceptualize therefore having many possible interpretations. The modeller implemented the decision function in ADM by adopting a traditional method of functional decomposition (p.54 [Dav96]).

---



**Example 4.8. Explicit meaning in SD.** The MUL artefacts constructed by software developers



**apply is**

**apply brake;**  
**generate opening.**

**release is**

**generate closing;**  
**release brake**

had an explicit meaning given by the relations between parts. These relations define parts of the model in terms of the parts of other models. There are also relations between parts of models and elements of the statement of requirements:

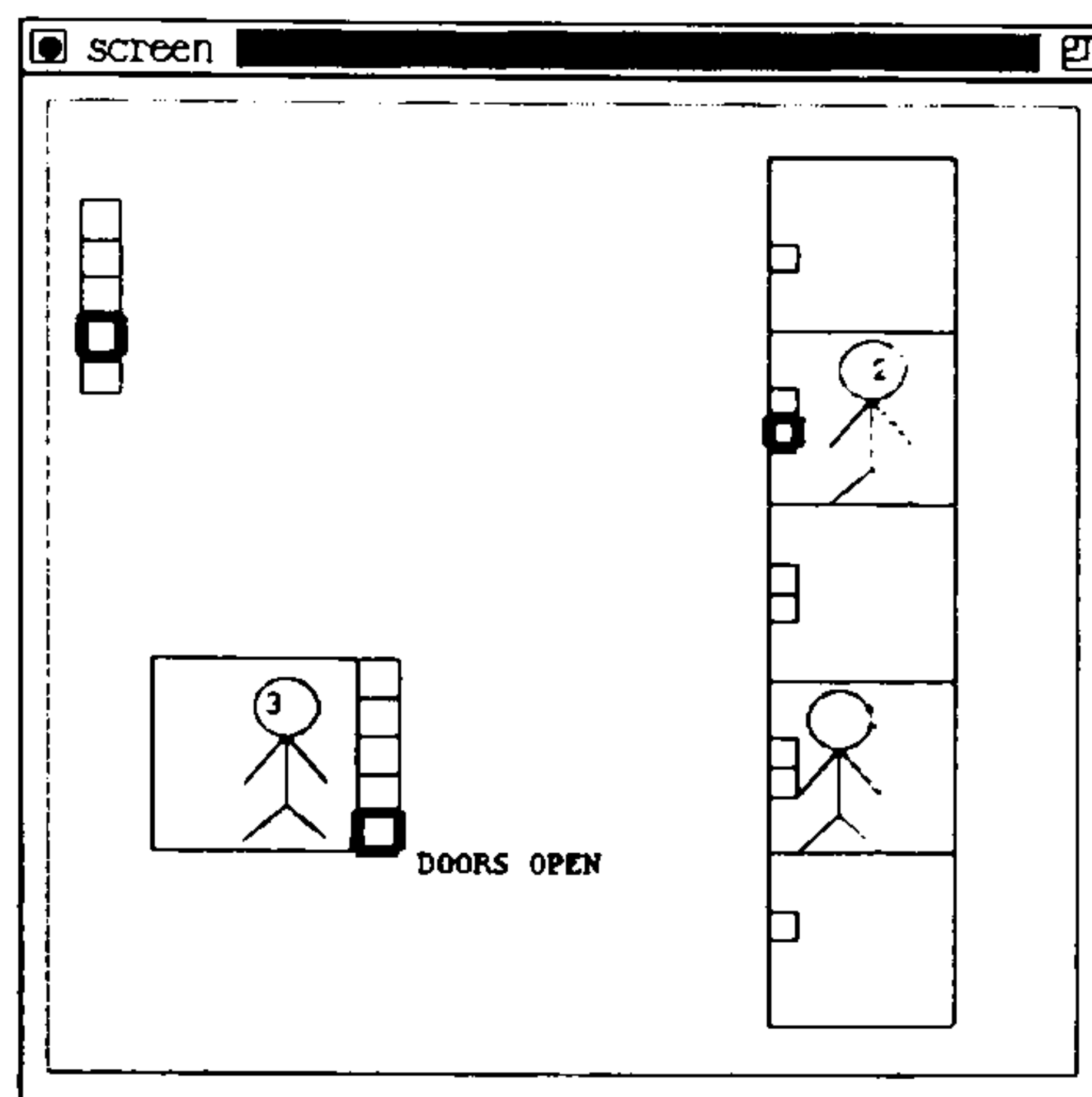
- The nouns “brake” and “shaft” are related to the object Class representations in the structure model.
- The verb phrase “the door is opened and closed by the brake” are related to the functional association between the brake and door Object classes in the structure model.
- The verb phrase “The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied” is related to the functional association between the brake and shaft Object classes in the structure model.
- The phrase “The brake is applied whenever the car arrives at a landing requested by a user” is related to the functional association between the brake and lift button Object classes in the structure model.
- The verbs “applied” and “released” are related to the actions and transitions of the brake Object class represented in the behaviour model.

This closed set of relations between the statement of requirements and models gave them an explicit meaning.



---

**Example 4.9. Implicit meaningfulness in EM.** It was found that the MUL visualization/animation



and LSD specification

```
agent landing(_F) {
state
  landButton
oracle
  floor direction brake
handle
  brake destination
protocol
  landButton[_F] == UP && _F == floor + 1 && brake == OFF -> brake = ON,
  landButton[_F] == DOWN && _F == floor - 1 && brake == OFF -> brake = ON,
  landButton[_F] != OFF && direction == NIL -> destination = _F,
  floor == _F -> landButton[_F] = OFF
}
```

captured a similar sense of meaningfulness as the subject. This suggests a direct correspondence between the elements of the artefacts and the elements of the subject in the mind of the modeller. In the case of the landing agent

- stating behaviour of the landing in terms of cause-and-effect,
- representing the landing as an agent which senses and responds to its environment, and
- concentrating on the observable aspects of the landing

all contribute to the intuitive meaning of the LSD specification.

---



---

**Example 4.10. Emergence and completeness in EM.** During the early stages of constructing the MUL artefacts there seemed no reason to change the SUL LSD specification of the user. However, after experimenting with the MUL visualization and animation it emerged that for a multiple user lift there had to be an up and a down button on each landing for the user to indicate his desired direction of travel. These features only emerged in the subject after the issue of request scheduling had to be addressed in the animation and ADM script. The protocol for pressing the button in the SUL

```
TRUE -> landButton{floor{_U}} = ON
```

was changed to

```
TRUE -> landButton{floor{_U}} = UP,  
TRUE -> landButton{floor{_U}} = DOWN
```

to include this emergent detail. It was realized that the change did not alter the essential meaning of the specification - the landing button being up or down still meant that it is was on - only the level of detail at which it represented the subject. This suggests an essential completeness about the LSD specification with respect to the subject.

Unexpected features were seen to emerge through interaction with visualizations and animations in most EM projects:

- it emerged during simulated sailing in the SBS that the model represented the boat capsizing even though this behaviour was not intentionally included by the modeller;
- the insufficient modelling of the synchronization between OXO players emerged when the computer-player played out-of-turn and won;
- it emerged during the railway simulation that the implementation of the guardsman protocol resulted in him stepping onto the track as the train was departing the station.

All these emergent features were identified and explored by the modellers resulting in subsequent modified models. This process resulted in a better understanding of the behavioural details of sailboats, playing OXO and railway systems.

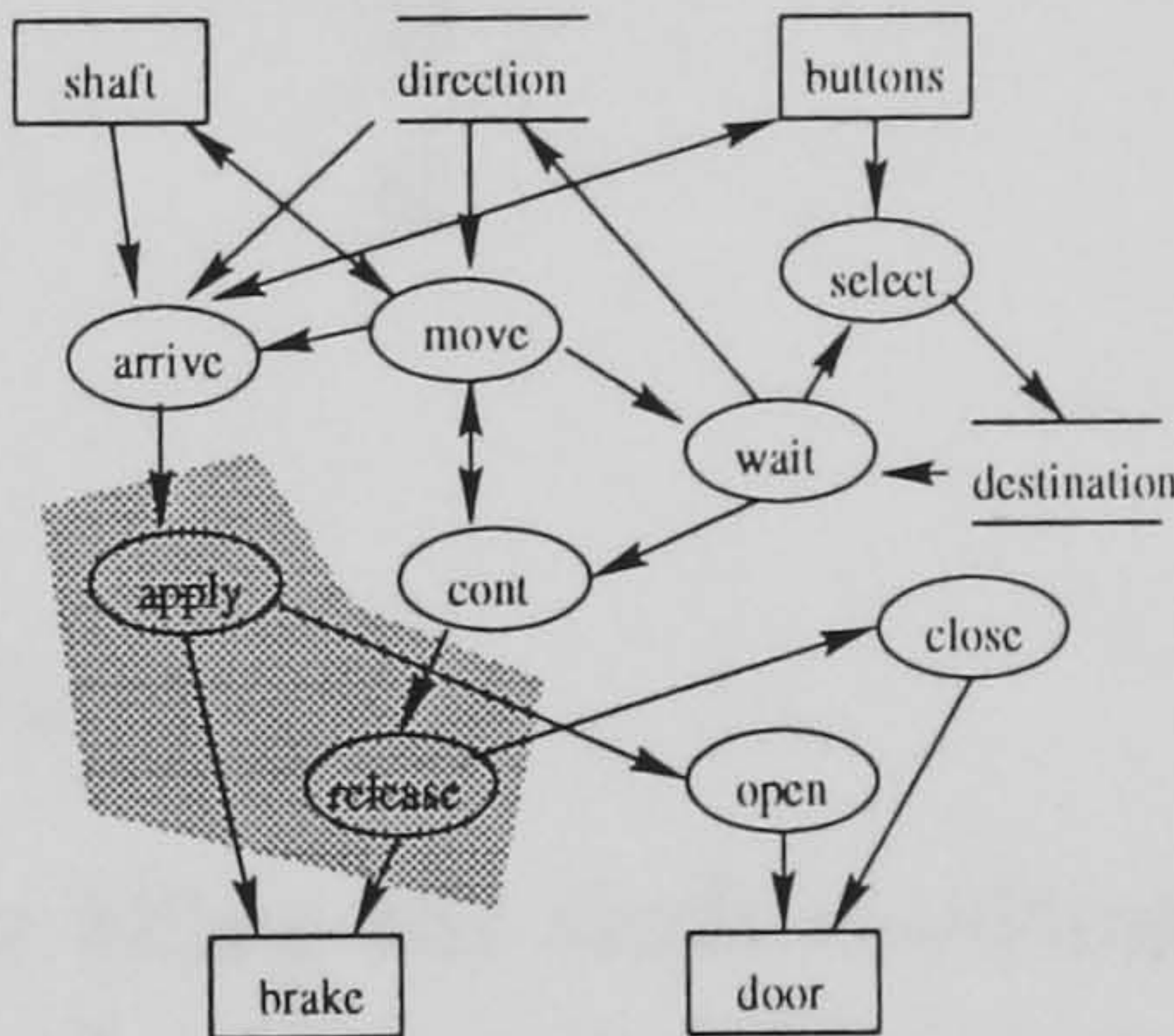
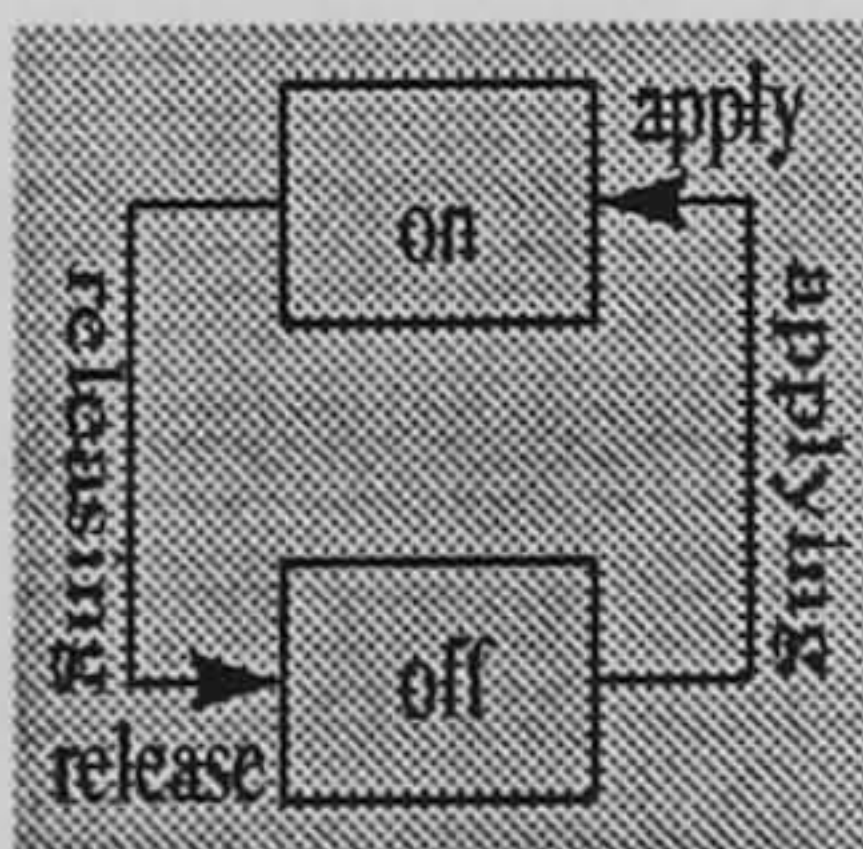
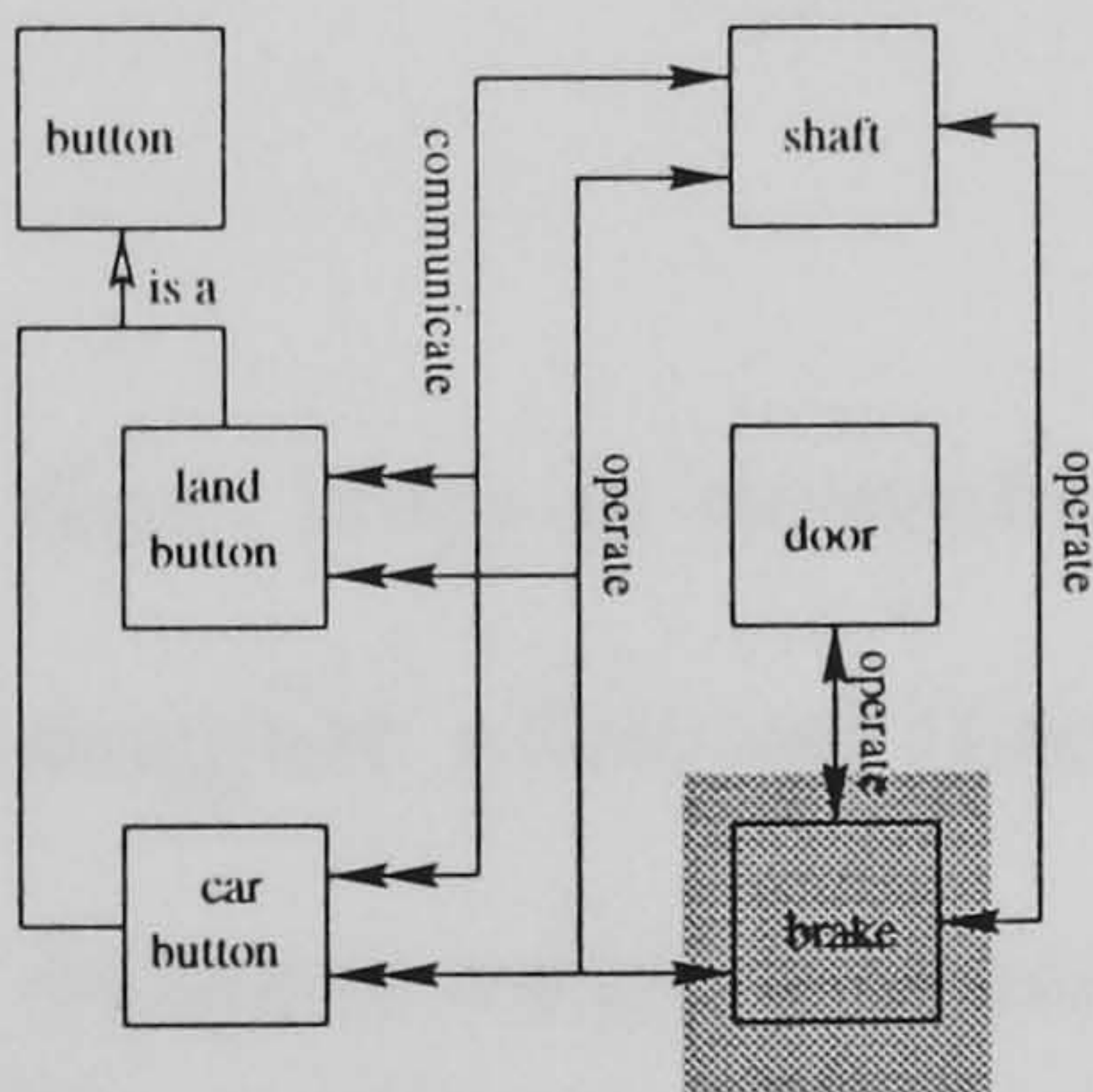
---



**Example 4.11. Completeness in SD.** The statement of requirements for the brake

... The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user ... The shaft mechanism releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

was found to be complete with respect to the MUL structure, behaviour and process models



apply is  
    apply brake;  
    generate opening.

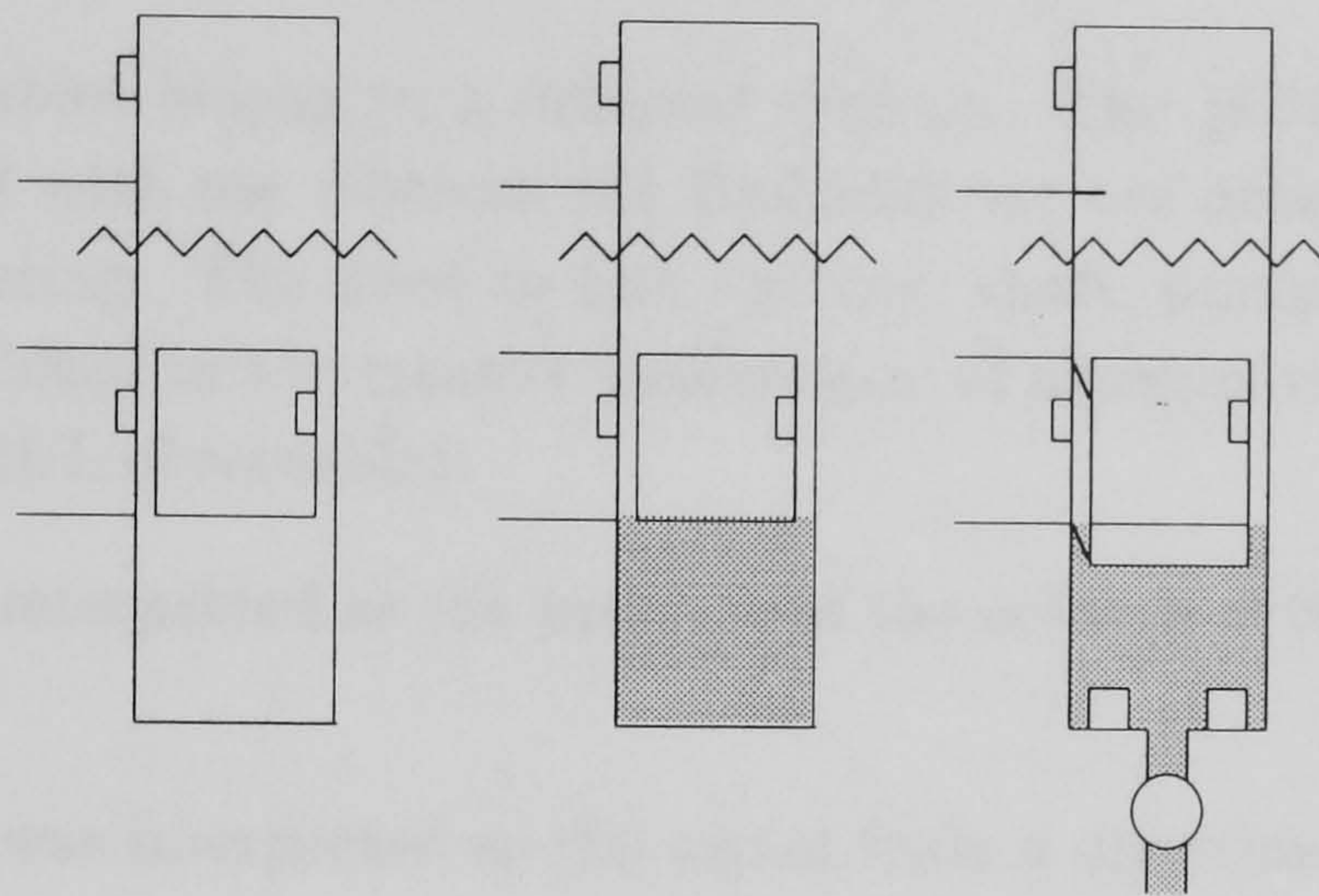
release is  
    generate closing;  
    release brake

It was found possible to check for inconsistencies in the description of structure and function in the statement, using the models, without having to think much about what the statement meant.



---

**Example 4.12. Incongruity and congruity in PD.** The following three sketches



show the three steps in designing the Hydrolift:

1. the designer retrieved their sketch for the MUL;
2. the designer added a representation of water filling the shaft resulting in the conflict between a conventional lift system and a hydraulic lift system;
3. the designer resolved the conflict between the two representations by adding more detail to the sketch in the form of appropriate components, such as a pump and sonar.

The design process was largely motivated by the designer's desire to progress from an incongruous sketch of a conventional lift system with water to a congruous sketch of a Hydrolift.

---



---

**Example 4.13. Incongruity and congruity in EM.** One source of conflict within an LSD specification is the lack of oracle-handle pairs. An oracle-handle pair indicates a link between agents. In the MUL the oracles and handles of the car and shaft agents are

**floor direction brake destination**

The modelling of the Hydrolift involved defining the pump, sonar and sensor agents with the following oracles and handles

**pressure chan1 chan2 direction sensed**

Each set of observables belong to a different domain. The MUL set are high-level concepts associated with use whereas the Hydrolift set are detailed concepts associated with engineering. The need to link the car, shaft, pump, sonar and sensor in the Hydrolift resulted in the creative exploration of alternative engineering interpretations of the MUL observables:

- the floor was interpreted as the pressure of the column of liquid at the base of the shaft;
- the direction was interpreted as the signal from a direction sensor;
- the destination was interpreted as a target pressure.

The changes resulting from these interpretations can be seen in comparing the MUL car and shaft agents with the pump agent in Appendix C. Such interpretations enabled the modeller to form oracle-handle pairs between agents to form a system.

The decision in the OXO project to define an umpire and board agents can be explained in terms of resolving conflicts due to observables. In the OXO model the LSD definition of the player

```
agent Player(P, 0) {
  state
    choice
  oracle
    turn Board
  handle
    Board
  protocol
    turn == P && available(Board, choice) -> take(Board, choice),
    !available(Board, choice) -> make_new_choice()
}
```

is incongruous to the modeller because of the observations of oracles without corresponding handles. This conflict was resolved in the OXO project by defining an umpire and board agents.

---



---

**Example 4.14. Divergence and convergence in EM.** The construction of the Hydrolift artefacts in EM involved divergence from and convergence towards the goal of a detailed animation.

An example of a divergent step was the juxtaposing of the MUL LSD specification with the LSD specification of a pump, sonar and direction sensor (see Example 4.13) which resulted in the creative exploration of alternative representations of the Hydrolift without much progress towards a detailed animation.

An example of a convergent step was the commitment to the LSD specification of the pump, shown in Example 4.13, and its transformation into the ADM entity

```
entity pump() {
  definition
    k = 100,
    change is (pressure < target) ? k :
              (pressure > target) ? -k : 0
  action
    target == pressure + change && brake == OFF -> brake = ON,
    change == 0 -> target = chan1*k,
    pressure == target -> chan2 = target/k,
    brake == OFF -> pressure = pressure + change,
    brake == ON && change != 0 -> brake = OFF
}
```

which formed a part of the final animation. It was found that the mixture of divergence and convergence was essential to the construction of the Hydrolift artefacts.

---

## Chapter 5

# Actions of EM, PD and SD

The activities of EM, PD and SD are essentially sequences of situated actions performed on artefacts by modellers, designers and software developers. Chapter 3 argued that the character of actions and other aspects of activities is determined by artefacts. It follows that the nature of the artefacts identified in Chapter 4 can be expected to determine the nature of actions.

This chapter compares the actions of EM, PD and SD to identify how they are essentially different. A suitable framework for comparison is provided by the theory of creative cognition [FWS92] that characterizes processes as generative and exploratory. The results of examining the processes of the lift project with respect to each kind of action are given. The characterization of processes is extended to observation and experimentation in scientific inquiry [Kap64].

### 5.1 Definition

Actions, in this thesis, correspond to the mental processes associated with the Geneplore model of Finke *et al* described in their book entitled “Creative Cognition: Theory, Research, and Application” [FWS92]:

The Geneplore model consists of two distinct processing components: a generative phase, followed by an exploratory phase (Figure 5.1). In the initial, generative phase, one constructs mental representations called preinventive structures, having various properties that promote creative



discovery. These properties are then exploited during an exploratory phase in which one seeks to interpret the preinventive structures in meaningful ways. These preinventive structures can be thought of as internal precursors to the final, externalized creative products and would be generated, regenerated, and modified throughout the course of creative exploration.

Appendix D gives a review of the book by Finke *et al* entitled “Creative Cognition: Theory, Research and Applications” [FWS92] and a critical analysis of the Geneplore model it describes.

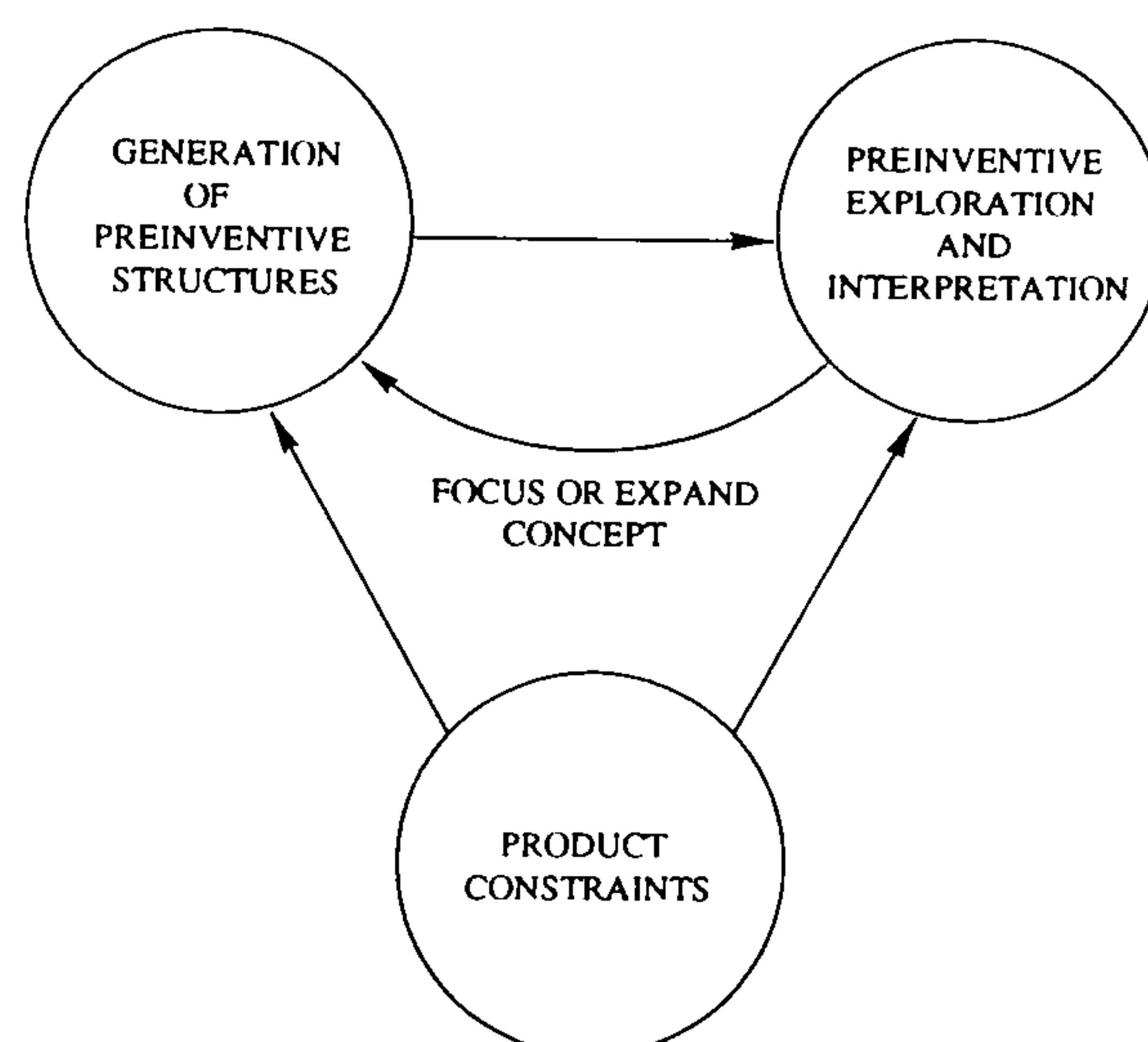


Figure 5.1: Geneplore Model.

This chapter investigates the application of generative and exploratory actions (Table 5.1) to artefacts with the properties that encourage analysis and creativity discussed in Chapter 4. Finke *et al* recognize that man’s “cognitive capacity” is limited and that external support is needed: “Although we are treating these structures as internal representations, there is no reason that the structures could not be externalized at any point in the creative act ... this has the advantage that one could then deal with more complex structures but the disadvantage that it might limit the flexibility in modifying and transforming the structures” [FWS92]. This chapter investigates the advantages and disadvantages of externalization. This chapter also addresses Norman’s claim that cognition can occur both in the head and in the world: “we should not see cognition as a purely unsupported activity” [Nor91].

Generative actions	Exploratory actions
Retrieval	Attribute finding
Association	Conceptual interpretation
Synthesis	Functional inference
Transformation	Contextual shifting
Transfer	Hypothesis testing
Reduction	Searching for limitations

Table 5.1: Generative and exploratory actions

## 5.2 Generative actions

This section examines the actions for generating artefacts that correspond to the six mental generative processes identified in [FWS92] (this book is reviewed in Appendix D) as being some of the most important for generating preinventive structures:

- the most basic processes consist of the **retrieval** of existing structures and the formation of **associations** among these structures. Typically these retrieval and associative processes happen quickly and automatically, but sometimes they are inhibited, resulting in mental blocks and fixation effects.
- a richer variety of structures results from the **synthesis** of component parts and by the **transformation** of the resulting forms. These processes usually yield more intricate creative possibilities than simple retrieval and association.
- **analogical transfer** is when a relationship or set of relationships in one context is transferred to another resulting in structures that are analogous to those that are already familiar. For example, early models of the structure of atoms resulted from analogical transfer of the relationships among the sun and planets in the solar system [FWS92].
- **categorical reduction** means mentally reducing objects or elements to more primitive categorical descriptions. For example, one might try to develop a better coffee cup not by considering it as a “cup” but as a container for keeping liquid hot and allowing it to be consumed [FWS92].



Finke *et al* [FWS92] recognize that these generative processes are not restricted to the generation of creative structures but are also used in the generation of structures for analysis: “Each of these generative processes has already been explored to some extent in traditional areas within cognitive psychology” [FWS92] with traditional cognitive psychology exploring the use of artefacts in essentially non-creative contexts. The use of generative actions by software developers in the lift project supports this view.

For convenience, the illustrative examples that go with this section have been organized into an appendix at the end of this chapter.

### 5.2.1 Retrieval

Constructing artefacts by retrieval was found to be a common technique in the lift project. By reusing parts of existing artefacts the modellers, designers and software developers were able to represent a subject more quickly than by synthesis. After establishing a primitive representation of the subject, by way of retrieval, the modellers, designers and software developers continued to refine the artefacts by using other generative actions.

Modellers began representing a subject by retrieving the artefacts describing the previous subject. This resulted in a continuity within EM artefacts, as shown in Example 5.1, with the artefacts of each subject being reused as the basis for constructing the artefacts of the next subject. For example, the LSD specification, visualization and animation of the MUL were reused by modellers as the starting point for constructing the Hydrolift artefacts.

The design of a subject began by retrieving sketches of the previous subject. As in EM, this resulted in a continuity within the sketches in the lift project. Sketches of subjects were reused as the basis for the sketching the next subject, as shown in Example 5.2. For example, the sketch of the MUL was used by designers as the starting point for sketching the Hydrolift.

Retrieval was used cautiously by software developers in the lift project. Inconsistencies were introduced into the structure, behaviour and process models by software developers reusing parts inappropriately. Retrieval was generally restricted



to a small number of parts corresponding to class definitions. These parts of the structure, behaviour and process model corresponded to parts of the statement of requirements that were similar for both subjects. For example, the software developer was able to reuse the MUL model of the brake in the Hydrolift models because the MUL and Hydrolift requirements for the brake were expressed by the same statement.

### 5.2.2 Association

Parts retrieved by modellers, designers and software developers were grouped to form new artefacts. Each part represented some part of the lift systems, such as a door, shaft or button. By associating these parts with one another they formed a representation of the subject as a whole.

EM artefact parts were associated without any explicit representation of how they were related. Relationships between parts were implied by the correspondence between the arrangement of parts in the subject and the arrangement of parts in the artefacts. For example, the LSD specification of the Hydrolift began as the combined MUL, pump, sonar and sensor LSD specifications, as shown in Example 5.4. Creative exploration of this incongruous association resulted in the eventual emergence of an LSD specification containing details about the pump, sonar and sensor agents. Sections of DoNaLD and ADM scripts were combined in a similar way to LSD specifications, without any explicit representation of how they were related, in order to construct visualizations and animations in the lift project, as shown in Example 5.4 .

Designers also used juxtaposing to associate artefact parts in the lift project. The designers arranged component caricatures into groups to represent the subject. For example, the sketch of the Hydrolift began as a representation of the MUL with a flooded shaft. Creative exploration of this incongruous association resulted in the emergence of the detailed Hydrolift sketch, as shown in Example 5.5.

In contrast to modellers and designers, software developers represented associations between artefact parts explicitly. For example, associations between classes in the structure model were represented as labeled arrows, as shown in Example



5.6. This avoided the problem of introducing creative properties into the models that might have resulted from simply arranging class definitions in the structure model without showing how they were related structurally and functionally. The software developers were able to transform the associations represented in the structure model into transitions and data-flows represented in the behaviour and process models by following the SD method of analysis, as shown in Example 5.6.

### 5.2.3 Synthesis

Artefact parts were created by the modellers, designers and software developers in the lift project whenever parts did not already exist. The SUL artefacts had to be synthesized because the SUL was the first subject in the lift project. The MUL and Hydrolift artefacts were constructed based on the SUL artefacts.

Synthesis of EM artefacts was found to follow a sequence in the lift project when the subject was novel. First the LSD specification was synthesized, shown in Example 5.7, followed by the synthesis of the visualization followed by the synthesis of the animation. The modeller represented his perception of the subject in an LSD specification then constructed the visualization and animation based on the description of observables and agents in the LSD specification. The modellers found it easier to start by creating an LSD description of the subject than to represent the structure and function of the subject in a visualization and animation directly. The LSD specification acted to guide the synthesis of the visualization and animation

The software developers had to verify synthesized models against the statement of requirements. The requirements were used to check synthesized parts of the structure, behaviour and process models, as shown in Example 5.8. The software developers preferred to transform the statement of requirements into models rather than synthesize and check models because it was more direct. Synthesized models typically needed a number of refinements before they satisfied the requirements, whereas models generated by transformation satisfied the requirements without any need for refinement or verification.



#### 5.2.4 Transformation

There were opportunities in EM and SD for constructing artefacts by transforming existing artefacts. Since transformations were perhaps the quickest way of generating artefacts they were favoured by modellers and software developers where appropriate.

Transformation was found to be the most common means of generating SD artefacts in the lift project. The software developer transformed the structure of the statement of requirements into the structure, behaviour and process models by following the method of analysis, as shown in Example 5.9. Details from previously constructed models were used in the transformations in addition to the statement of requirements.

The modeller performed a straightforward transformation from the structure of the LSD specification into an ADM script to construct animations in the lift project, as shown in Example 5.10. Essentially, the transformation was done by renaming agents as entities, privileges as definitions and protocols as actions.

#### 5.2.5 Analogical transfer

Although analogical transfer is normally associated with the relation between mental models [HT95], an externalized analogical transfer was observed in the lift project. Analogical transfer is when relations in one context are transferred to another in order to generate structures that are analogous to those that are already familiar [FWS92]. In order to avoid confusion I shall simply refer to the externalized form of analogical transfer as “transfer.” Transfer in the lift project involved modellers, designers and software developers generating new artefacts by keeping the structure of existing artefacts and changing their content.

Modellers often retrieved artefacts to reuse their structure in the lift project. For example, the modeller retrieved the shaft agent definition and reused its structure to construct the pump in the Hydrolift, as shown in Example 5.11. Transfer was also used by modellers in order to construct visualizations and animations: the organization of DoNaLD and ADM scripts made it possible to redefine shapes, whilst keeping their relative positions the same, and make changes to the behaviour



of entities, whilst keeping their basic agency and protocols the same. In this way, the essential structure and function of lift systems, established in the early stages of the lift project with the SUL, was preserved throughout.

Designers in the lift project performed transfer between sketches. Transfer was achieved by reusing the layout of existing sketches. New component caricatures were added to complete the sketch after transfer, as shown in Example 5.12. The detail of the sketch was changed whilst preserving the layout that was established early in the lift project with the sketch of the SUL.

Transfer was seldom used for generating SD artefacts. The reason for this was that the meaning of SD models was determined more by the arrangement of symbols than by the symbols themselves. By transferring the structure of a model the greater part of its meaning was transferred with it, as shown in Example 5.13. Reusing the structure of artefacts had to be done with caution by software developers so as to avoid introducing inconsistencies and other creative properties into the models.

### 5.2.6 Categorical reduction

As with analogical transfer, categorical reduction is normally associated with mental structures. Categorical reduction means mentally reducing objects to more primitive descriptions of constituent parts by disregarding their more abstract higher-level conceptual structure [FWS92]. However, an externalized categorical reduction was observed during EM, PD and SD in the lift project. So as to avoid confusion I shall refer to the externalized categorical reduction simply as “reduction”. Reduction in the lift project involved modellers and software developers generating new artefacts by stripping away the higher-level structure of existing artefacts.

Reduction was commonly used by modellers for generating artefacts in the lift project, as shown in Example 5.14. New artefacts were generated quickly by retrieval and association. Modelling continued by repeatedly reducing the artefacts into basic parts and rearranging the parts to form new artefacts. In this way, the modeller gradually converged upon a detailed representation of the subject. The structure of the EM artefacts was found to support the decomposition of artefacts into meaningful parts right down to the most basic element - the observable.

Reduction was found by software developers to be limited as a technique for SD. The importance of structure to the meaning of the models made it difficult to reduce SD models without rendering them meaningless, as shown in Example 5.15. Reduction was not performed on the structure, behaviour and process models because the parts, including states, transitions and actions, tended to have little meaning without the context of the symbols that surrounded them. Reduction was done by identifying a class definition then separating it from its context by representing its relationship to other classes as an interface.

### 5.3 Exploratory actions

This section examines the actions for exploring artefacts that correspond to the six mental generative processes identified in [FWS92] (this book is reviewed in Appendix D) as being some of the most important for creative exploration:

- **attribute finding** is the systematic search for emergent features in the structures.
- **conceptual interpretation** refers quite broadly to the process of taking a structure and finding an abstract, metaphorical, or theoretical interpretation of it. Conceptual interpretation can be thought of as the application of world knowledge or naive theories to the task of creative exploration.
- **functional inference** refers to the process of exploring the potential uses or functions of a structure. This process is often facilitated by imagining oneself actually trying to use the object in various ways.
- **contextual shifting** is considering a structure in new or different contexts as a way of gaining insights about other possible uses or meanings of the structure. This process often helps to overcome fixation effects and other obstacles to creative discovery.
- **hypothetical testing** is where one seeks to interpret the structures as representing possible solutions to a problem. A creative solution to a problem can often be found when more direct methods fail.



- **searching for limitations** is searching out what structures will not work and are not feasible. This is often just as important as actually discovering what will work.

The following examination focuses on the use of exploratory actions in EM and PD because there was little evidence of software developers using exploratory actions in the lift project.

For convenience, the illustrative examples that go with this section have been organized into an appendix at the end of this chapter.

### 5.3.1 Creative exploration and SD

There was little evidence of software developers using exploratory actions in the lift project. This was probably due to the analytical properties of the artefacts of SD discussed in Chapter 4:

- Familiarity
- Unambiguity
- Explicit meaning
- Completeness
- Consistency
- Convergence

These properties mean there is little incentive for the software developer to explore the artefacts, as shown in Example 5.16. There is no incentive because the artefacts make finding emergent features difficult and unnecessary to explore:

- the property of completeness means that all the information the software developer needs is within the artefacts so there is little incentive to search for emergent features;
- the SD method maps symbols from one domain to symbols in another so there is little incentive for the software developer to explore alternative interpretations of the subject;

- the functional meaning of the subject is explicit in the artefacts so there is little incentive for the software developer to explore alternative behaviours;
- the property of completeness means that the artefacts have the same formal meaning independent of context so there is little incentive for the software developer to shift the context of the artefacts either physically or conceptually;
- the property of convergence means that the SD process converges upon a solution so there is little incentive for the software developer to test artefacts;
- so long as the software developer follows the SD method the only limitations of the artefacts will be due to limitations in the statement of requirements which is not his responsibility.

SD in the lift project was essentially a methodical transformation of the statement of requirements into the structure, behaviour and process models. This suggests that SD is essentially a generative activity with little incentive for creative exploration.

### 5.3.2 Attribute finding

Modellers found emergent features in visualizations and animations. These features were not intentionally modelled and were only found by exploring the EM artefacts. The features discovered by modellers were included in subsequent visualizations and animations. Discoveries about observables and agency in the subject informed revisions to LSD specifications [BR94], as shown in Example 5.17.

Finding attributes in sketches was limited by the designers' knowledge of lift systems. The success of thought experiments in discovering emergent features in the sketches depended on the knowledge of the designers [Gre70]. The designers were not experienced in the lift project so they were unable to find many structural and functional attributes within the sketches. Experienced designers would have been expected to benefit far more from exploration based upon their mental models of lift systems.



### 5.3.3 Conceptual interpretation

Modellers and designers explored the subject by interpreting it in terms of abstract and concrete concepts. The modellers interpreted the SUL, MUL and Hydrolift in terms of the concepts of LSD, DoNaLD and ADM. The designers interpreted the SUL, MUL and Hydrolift in terms of the elements of a “graphical language” in the sense of Ferguson [Fer92].

Modellers interpreted the subject in terms of the LSD concepts of agent, protocol, derivate and observable. By interpreting the subject in terms of these concepts the modellers were able to represent and explore the SUL, MUL and Hydrolift in familiar terms that corresponded to elements that they perceived within the subject.

The modellers interpreted the LSD specification in terms of DoNaLD and ADM concepts in order to construct visualizations and animations in the lift project, as shown in Example 5.18. The concepts of these languages have precise and unambiguous meanings that define the structure and function of the visualizations and animations. Interpreting the LSD specification in terms of these concepts resulted in the emergence of features in the subject to do with the geometry of shapes and the synchronization of actions.

Designers interpreted the subject in terms of the elements of a “graphical language” . Although designers did not use a verbal language, they did use conventions for representing subjects in sketches. These conventions can be thought of as a kind of “graphical language” in the sense of Ferguson [Fer92]. The designers explored the spatial relations between components in the subject by arranging component caricatures in the sketch. The patterns of caricatures in a sketch can be thought of as “statements” that describe the subject in a graphical language of the designer, as shown in Example 5.19.

### 5.3.4 Functional inference

Modellers and designers used artefacts to explore the emergent behaviour of the subject. Norman has observed that mental models of systems are difficult to “run” [Nor91] so interactive artefacts play a particularly important supportive role in the



process of functional inference. The visualization and animation were found to be more suitable for functional inference than the LSD specification and sketch.

Visualizations and animations were found to be most appropriate for supporting functional inference in the lift project. Visualizations and animations provided the modeller with a context for interaction that mimicked that provided by the subject. For example, the MUL visualization and animation represented lift buttons by graphics that were sensitive to the point-and-click of a mouse and represented the state of the lift system by a picture for the modeller to see that reflected the current state of the lift system, as shown in Example 5.20. The ADM entities simulating LSD agents meant that the animation was more realistic than the visualization. However, by the modeller playing the roles of LSD agents, the visualization was found to give more freedom in inferring alternative functions of the subject.

Functional inference based on the LSD specification and sketch was found to be limited as a technique for exploration. Functional inference using LSD specifications and sketches involved thought experiments [Kap64] that depended on the knowledge and experience of lift systems. The modellers and designers in the lift project were not experienced in the workings of lift systems. Designers who have experience of lift system components are able to infer more complex behaviours from design sketches.

Modellers and designers explored the function of the subject by imagining themselves using the lift system. Modellers commonly represent themselves as agents in LSD specifications to help them imagine their interaction with a system, as shown in Example 5.20. This accords with the findings of Finke *et al* who state that “the process of functional inference is often facilitated by imagining oneself actually trying to use the object in various ways” [FWS92].

### 5.3.5 Contextual shifting

The artefacts of EM and PD were found to be context sensitive which meant that they were suitable for supporting creative discovery through contextual shifting. New features emerged in the artefact by placing it in different contexts. Modellers and designers incorporated the emergent features in subsequent artefacts.



Contextual shifting was used by modellers to explore EM artefacts. Modellers changed the context of the LSD specification, visualization and animation:

- *conceptual shifting* involved changing the context of an agent definition within an LSD specification, as shown in Example 5.21;
- *physical shifting* involved changing the physical environment of the computer running the visualization or animation.

Physical contextual shifting was limited in the lift project to changing the person interacting with the visualization or animation. In the future it is hoped that interfaces will be developed that allow components and systems to be linked to the computer so that they interact directly to change variables in the visualizations and animations. These physical devices could then replace their virtual representations in the form of ADM entities over a period of systems development.

Contextual shifting by designers had some similarities with conceptual context shifting by modellers in the lift project. Contextual shifting by designers involved changing the context of a component caricature within a sketch rather like changing the context of an agent definition within an LSD specification. This typically had the effect of changing the functional meaning of the component representation.

### 5.3.6 Hypothesis testing

The principal aim of the modellers and designers was to construct satisfactory representations of the subject. The artefacts were searched to find similarities that confirmed them as being appropriate representations of the subject. The more similarities the modellers and designers discovered the more confident they were that the artefacts were indeed satisfactory representations of the subject.

Modellers applied hypothesis testing to visualizations and animations. Testing of the LSD specification was less common because its direct correspondence to the subject as perceived by the modeller generally meant it was a satisfactory representation, albeit one that lacked structural and functional detail. The visualizations and animations were regularly tested by modellers to check their representation of

the subject's structure and function. For example, the visualization and animation were regularly tested for essential safety and liveness properties [MP92a]: the car must never move when the door is open [She96]; all user requests must be serviced within a respectable time period.

It was recognized during the lift project that there were alternative approaches to testing for safety and liveness properties than having the modeller observe the visualization and animation:

- Define an hypothesis testing ADM entity that would automatically check the constraints and produce a signal if they were violated. This idea raised questions about the reliability of the observations of such an agent.
- Use formal methods for verifying concurrent real-time systems [OG75, Bar85, Pnu86, Hoo91, AO91, MP92a] to analyze the ADM script. This idea raised the issue of how appropriate it was to attempt to formalize the behaviour of a visualization or animation.

These two approaches correspond to “watchdogs” and temporal logic as recommended for system verification by Harel [Har92].

Hypothesis testing in PD was found to be rather limited. Designers did not have anything equivalent to the visualization and animation with which to evaluate sketches. The designers lacked the experience needed to generate mental models of sufficient detail to test for safety and liveness properties.

### 5.3.7 Searching for limitations

Modellers and designers searched for inadequacies in artefacts. Instead of searching for examples of similarities between subjects and artefacts the modellers and designers purposely searched for counter-examples that showed mismatches between the subject and its representation. This was to counteract the natural tendency of the modellers and designers to find evidence that confirmed the artefacts as satisfactory. This phenomenon is known as “confirmation bias” [FWS92, Wol92].

Modellers searched for limitations in visualizations and animations by setting up scenarios that might potentially fail safety and liveness criteria. For example,



users were placed on the top and bottom floors in the MUL and Hydrolift animations in order to try and fail the liveness property that all user requests must be serviced within a respectable period of time. By setting up scenarios the modellers were able to search for limitations in the behaviour of visualizations and animations.

Searching for limitations was found to be a less effective technique in PD than in EM. Designers did not have anything equivalent to the visualization and animation with which to setup scenarios for determining the limitations of designs. Designers lacked the knowledge and experience needed to generate mental models of sufficient detail to test routine, let alone, exceptional lift system behaviour.

## 5.4 Further characterizations of actions

In this section, observation, experimentation, method and methodology, in the sense of Kaplan [Kap64], are discussed in terms of generative and exploratory actions.

### 5.4.1 Observation and experimentation

In [Kap64] Kaplan describes various kinds of experiments for the purposes of scientific enquiry:

- Methodological experiments serve to develop or improve a technique of scientific enquiry.
- Heuristic experiments are designed to generate novel ideas for further scientific enquiry and are of the form “What would happen if ...”
- Fact-finding experiments aim at determining some particular magnitude or property of a relatively familiar object or situation.
- Boundary experiments are fact-finding experiments to determine the extent of a theory or law.
- Simulation experiments are designed to learn what will happen in artificial conditions which directly correspond to real ones.

- Nomological experiments aim to establish a theory by confirming or disproving an hypothesis.
- Illustrative experiments do not add anything to the knowledge about something only to the knowledge of the audience.
- Thought experiments are those involving mental concepts as opposed to physical apparatus and of the form “Imagine what if ...”

There are clearly parallels to be drawn between these kinds of experiments and the exploratory acts discussed previously in this chapter:

- The purpose of an heuristic experiment is to interpret an object or situation in a creative way similar to the act of creative exploration.
- Fact-finding experiments correspond to exploring an artefact to discover emergent features in creative exploration.
- Models are used in simulation experiments in the same way models are used to explore their function and shift contexts in creative exploration.
- Nomological experiments correspond to the creative process of hypothetical testing.
- Thought experiments correspond to the mental processes described by Finke *et al* [FWS92] whereas the experiments using physical apparatus correspond to the actions described previously in this chapter.

This suggests that the scientific knowledge generated through experimentation is the result of creativity as well as analysis.

Although experimentation might be creative it could be argued that observation is a passive non-creative activity resulting in the recording of facts during an experiment. However, Kaplan [Kap64] counters this argument with the following statement:

Basically, experimentation is a process of observation, to be carried out in a situation especially brought about for that purpose ... No scientific



observation is wholly passive; how much the scientist intervenes before or during the process of observation is a matter of degree. Correspondingly, there is no sharp distinction between observation and experiment, only a series of gradations and intermediates.

This would suggest that both observation and experiment involve creativity to some degree resulting in certain artificiality in scientific knowledge.

This theme of perceived reality being to some extent the products of the experimenter's creativity is continued by Gooding in [Goo90] in his discussion on construals (see Section 4.4.1):

“Making sense” involves achieving stable interaction with a bit of the world. If a construal succeeds in this, then it will be accepted provisionally as a model of the phenomenon ... The effectiveness of a construal emerges as it is vindicated in the outcomes of further exploratory and communicative behaviour. After a while it becomes “easy to see” phenomena in terms of it, and it paves the way for the “self-evidence.”

It is clear from the above statement that he views observation as an activity which is combined with the creation, by the experimenter, of the construal. The same sentiment is shown by others writing about science [Bro86].

### 5.4.2 Methods and methodology

In [Kap64] Kaplan defines a method as a general technique in science:

- Forming concepts and hypotheses.
- Making observations and measurements.
- Performing experiments.
- Building models and theories.
- Providing explanations.
- Making predictions.

There are clearly parallels between these examples and the generative and exploratory actions discussed previously in this chapter. This suggests that the actions in this chapter can be thought of as methods in the sense of Kaplan [Kap64].

Kaplan [Kap64] is careful to distinguish between the terms method and methodology - two terms that are often confused especially in the area of SD [You92]. He defines methodology as

the study - the description, the explanation, and the justification - of methods, and not the methods themselves ... The aim of methodology, then, is to describe and analyze these methods, throwing light on their limitations and resources, relating their potentialities to the twilight zone at the frontiers of knowledge. It is to venture generalizations from the success of particular techniques, suggesting new applications, and to unfold the specific bearings of logical and metaphysical principles on concrete problems, suggesting new formulations. It is to invite speculation from science and practicality from philosophy. In sum, the aim of methodology is to help us to *understand*, in the broadest possible terms, not the products of scientific inquiry but the process itself. [Kap64]

In this sense, the examination of generative and exploratory actions in this chapter may be construed as a methodology. However, it should be noted that the purpose of this examination is to gain a better understanding of the activities of EM, PD and SD not to improve upon them. Kaplan [Kap64] warns against attempting to refine methods through retrospective reconstruction, arguing that “pressing methodological norms too far we may inhibit bold and imaginative adventures of ideas.” Perhaps, a contributing factor to the analytical nature of SD methods, such as the Shlaer-Mellor object-oriented method of analysis and design [SM88, SM92], is due to an emphasis on methodology in an effort to solve the software crisis.

## 5.5 Summary

In this chapter the actions of EM, PD and SD were compared. It was found that actions of both modellers and designers in the lift project consisted of generative



and exploratory actions:

- retrieval of artefacts from archives and association through combination;
- synthesis by essentially reassembling and rearranging artefact parts;
- transfer of high-level structure between artefacts;
- reduction of artefacts into constituent meaningful parts.

It was found in the lift project that SD had similar generative actions but that they were more constrained and transformational than in EM and PD. There was less incentive for the software developer to explore the resulting artefacts. Exploratory actions were found to be important to the construction of artefacts in EM and PD:

- searching for emergent features in the artefact;
- interpreting the artefact in terms of abstract concepts;
- inferring the function of the artefact;
- shifting the context of the artefacts;
- testing the artefact as a solution to a problem;
- searching for limitations in artefacts.

These exploratory actions were found to be done in parallel with exploring the subject.

This chapter also shows similarities between experimental approaches in science [Kap64] and EM, PD and SD in the lift project. The discussion of methodology in science [Kap64] gives insight into the use of methodical approaches in SD.

## Appendix: Illustrative examples for Sections 5.2 and 5.3

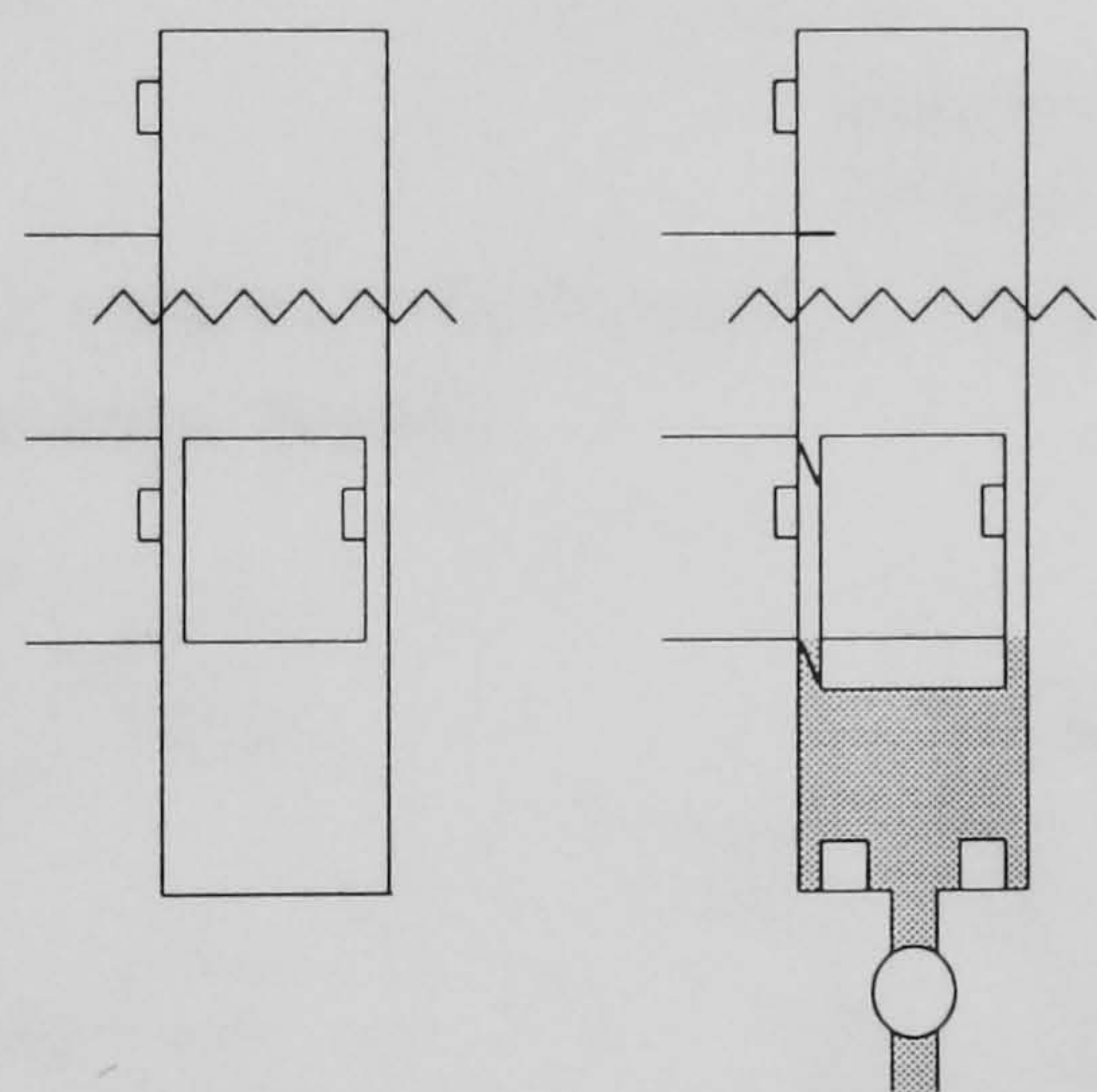
**Example 5.1. Retrieval in EM.** The outline of the MUL LSD specification (shown on the left) and the outline of the Hydrolift LSD specification (shown on the right) show that retrieval led to a continuity in the basic structure of the LSD specifications.

<pre> agent door() { state   door oracle   brake } </pre>	<pre> agent door() { state   door oracle   brake } </pre>
<pre> agent landing(_F) { state   landButton oracle   floor direction brake handle   brake destination } </pre>	<pre> agent landing(_F) { state   landButton oracle   sensed brake handle   brake } </pre>
<pre> agent car(_F) { state   carButton oracle   floor direction brake handle   brake destination } </pre>	<pre> agent car(_F) { state   carButton oracle   chan2 handle   chan1 } </pre>
<pre> agent shaft() { state   floor destination direction oracle   brake handle   brake } </pre>	<pre> agent pump() { state   change target oracle   brake pressure chan1 handle   brake pressure chan2 } </pre>

The change in agent name, from shaft to pump, and the changes in observables reflects the refinement of the Hydrolift specification following the initial generation by retrieval.



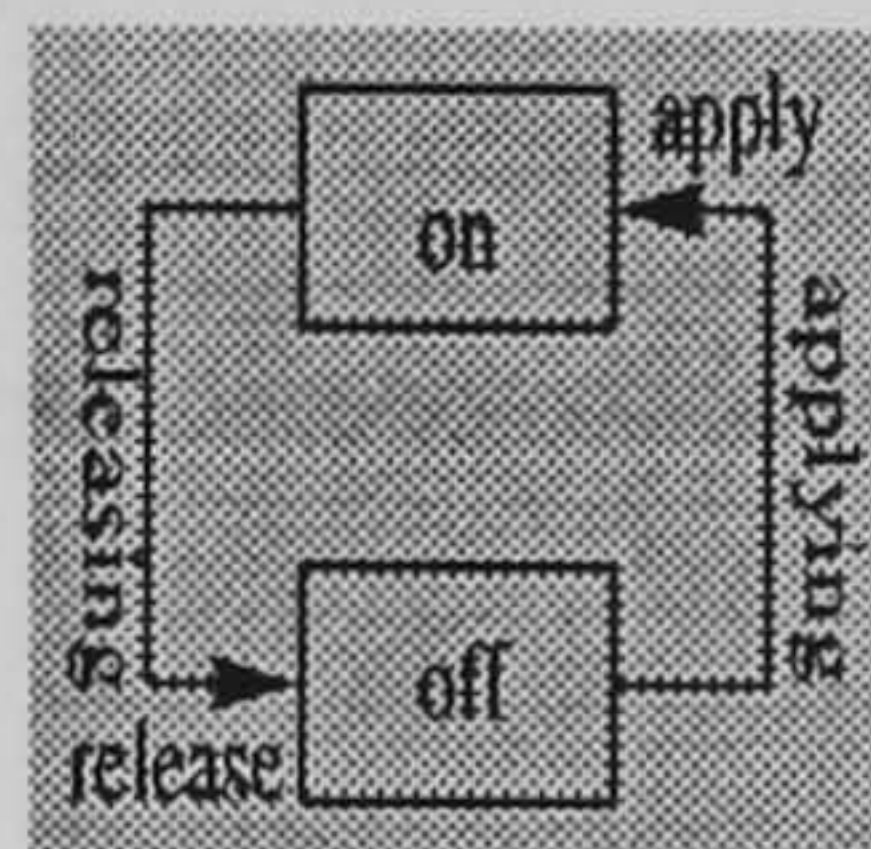
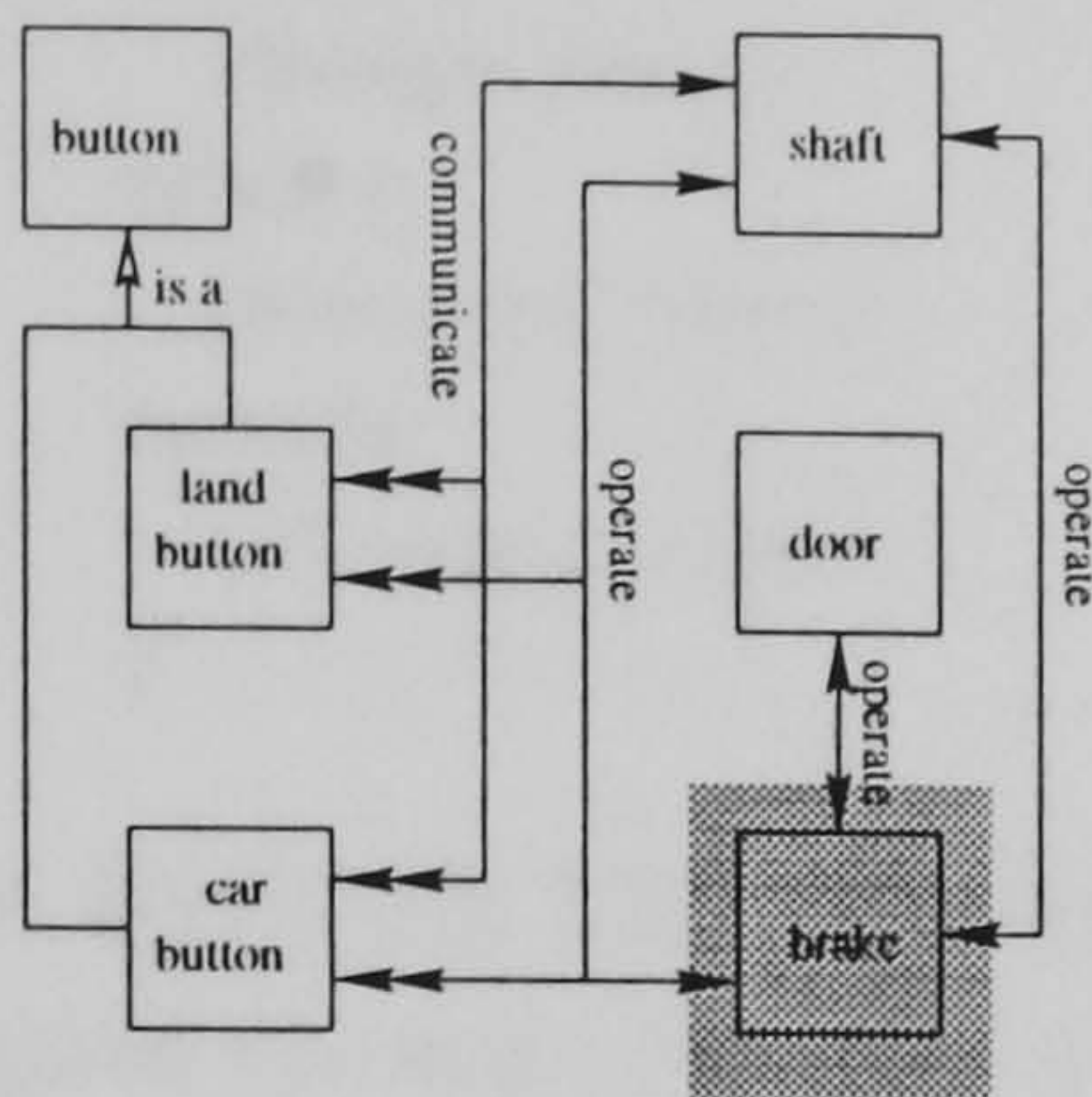
**Example 5.2. Retrieval in PD.** The sketches of the MUL and Hydrolift show that retrieval led to the continuity of the basic structure in lift systems during PD in the lift project.



Additional component representations were added during the refinement of the Hydrolift following its initial generation by retrieval.



**Example 5.3. Retrieval in SD.** The MUL artefact parts corresponding to the brake Object class (shown highlighted)



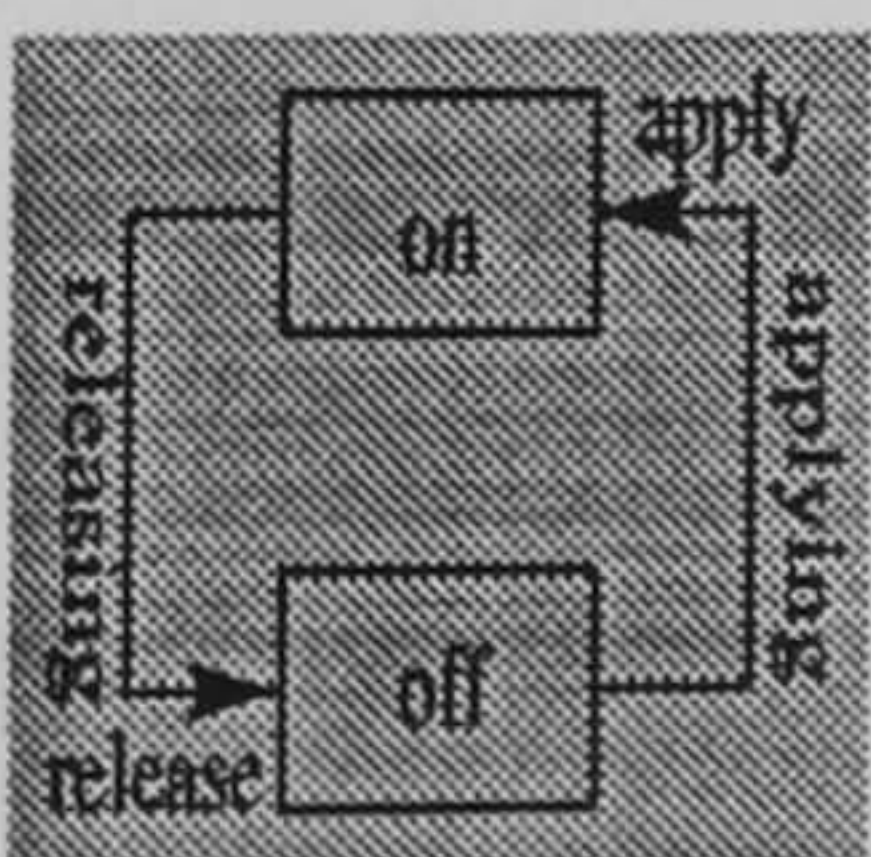
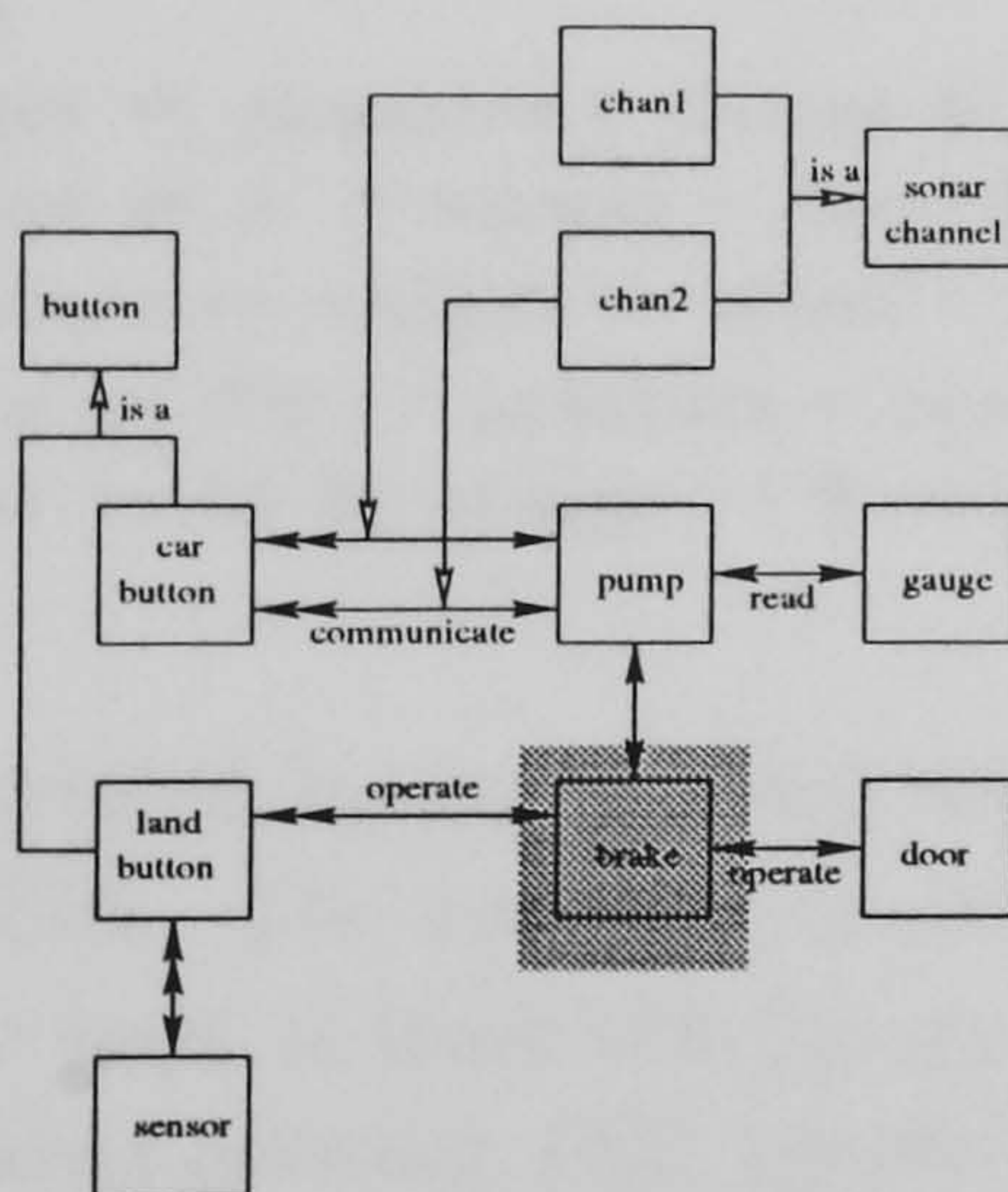
apply is

apply brake;  
generate opening.

release is

generate closing;  
release brake

were retrieved and reused by software developers in the generation of the Hydrolift structure, behaviour and process models



apply is

apply brake;  
generate opening.

release is

generate closing;  
release brake

As can be seen from the models, the Hydrolift brake is related to the car button and door the same as with the MUL brake. Also, the relation between the MUL brake and shaft is the same as the relation between the Hydrolift brake and pump.



**Example 5.4. Association in EM.** The association between the Hydrolift pump, sonar and sensor was represented by juxtaposing their agent definitions

<code>agent pump() {</code>	<code>agent sonar() {</code>	<code>agent sensor()</code>
<code>  state</code>	<code>  oracle</code>	<code>  state</code>
<code>    change target</code>	<code>    chan1</code>	<code>    floor</code>
<code>  oracle</code>	<code>  handle</code>	<code>  oracle</code>
<code>    pressure chan1</code>	<code>    chan2</code>	<code>    direction</code>
<code>  handle</code>	<code>}</code>	<code>  handle</code>
<code>    pressure chan2</code>		<code>    sensed</code>
<code>}</code>		<code>}</code>

within the Hydrolift LSD specification. The LSD specification shows that the modeller imagines the associations to be three distinct kinds of agent within the Hydrolift that share certain observables. The LSD specification does not describe the detailed structure or function of components.

Within the framework of the ADM, the entities corresponding to the above agents, such as the pump entity

```
entity pump() {
  definition
    k = 100,
    change is (pressure < target) ? k :
              (pressure > target) ? -k : 0
  action
    target == pressure + change && brake == OFF -> brake = ON,
    change == 0 -> target = chan1*k,
    pressure == target -> chan2 = target/k,
    brake == OFF -> pressure = pressure + change,
    brake == ON && change != 0 -> brake = OFF
}
```

are associated by mechanisms for the instantiation of entities and the evaluation of variables. The associations between entities within the ADM are formalizable. A major part of transforming an LSD specification into scripts is interpreting the associations between LSD agents in terms of structure and function.

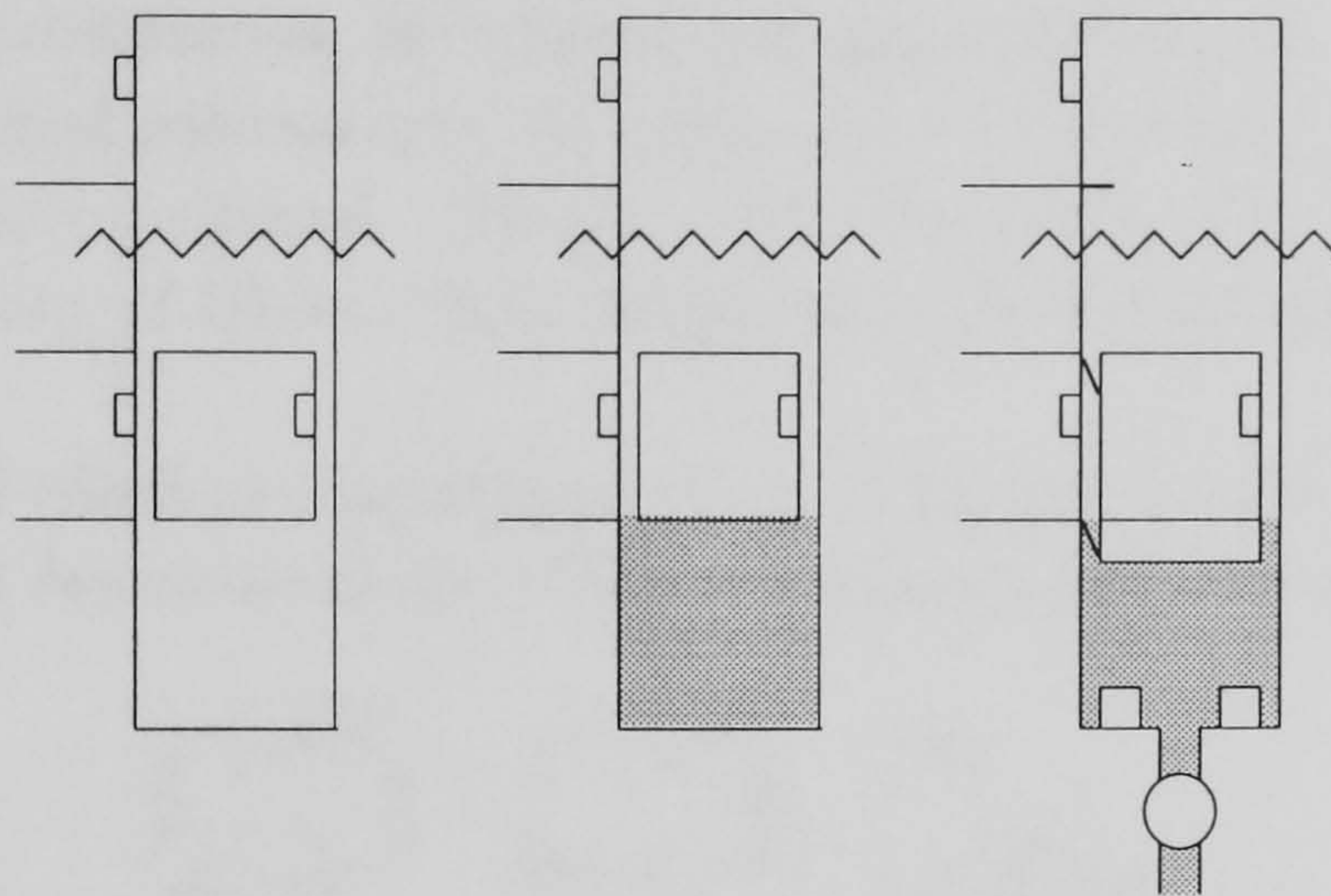
Most EM projects have involved the modellers associating agents by juxtaposing definitions in an LSD specification and then transforming the LSD specification into a script defining structural and functional relations. For example, the classroom simulation project can be thought of as a two-tier process:

- associate pupils and teachers by describing them in an LSD specification with shared observables;
- formalize the associations between pupils and teachers by transforming the LSD specification into scripts, such as the decision function into ADM.

A similar two-tier process was observed in other EM projects, including the VCCS, OXO and SBS.



**Example 5.5. Association in PD.** During the early stages of PD in the lift project the designer associated components by juxtaposing representations of them within a sketch. This was rather similar to the juxtaposing of agents within an LSD specification during EM. Subsequent stages of PD involved the designer exploring these associations. For example, the following three sketches



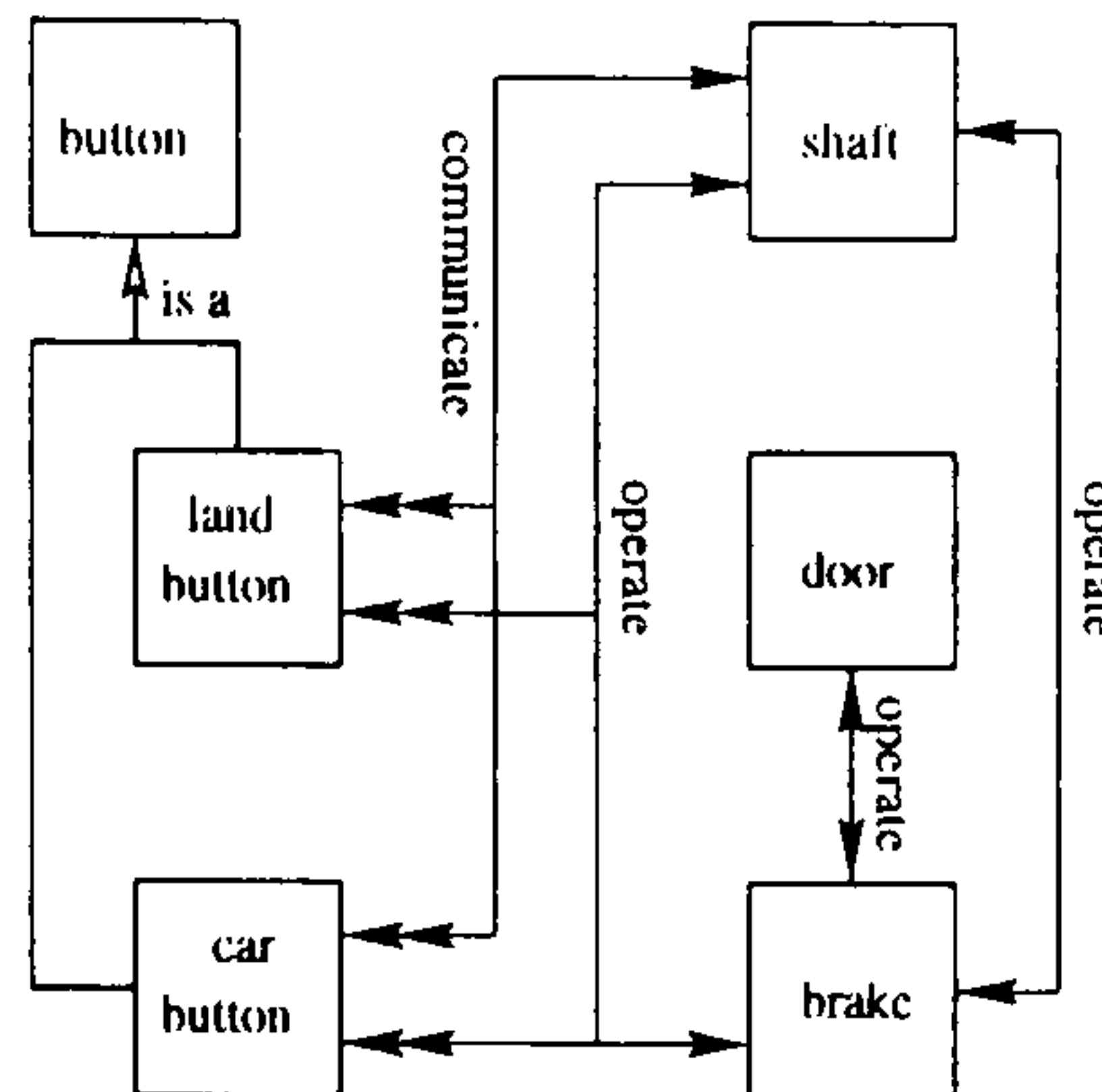
show the three steps in designing the Hydrolift:

1. the designer retrieved the sketch for the MUL;
2. the designer formed an incongruous association between a conventional lift system and water;
3. during the exploration of the association the designer added devices including a pump and sonar device.

The design process was largely motivated by the designer's desire to progress from an incongruous sketch associating a conventional lift system with water to a congruous sketch of a Hydrolift.

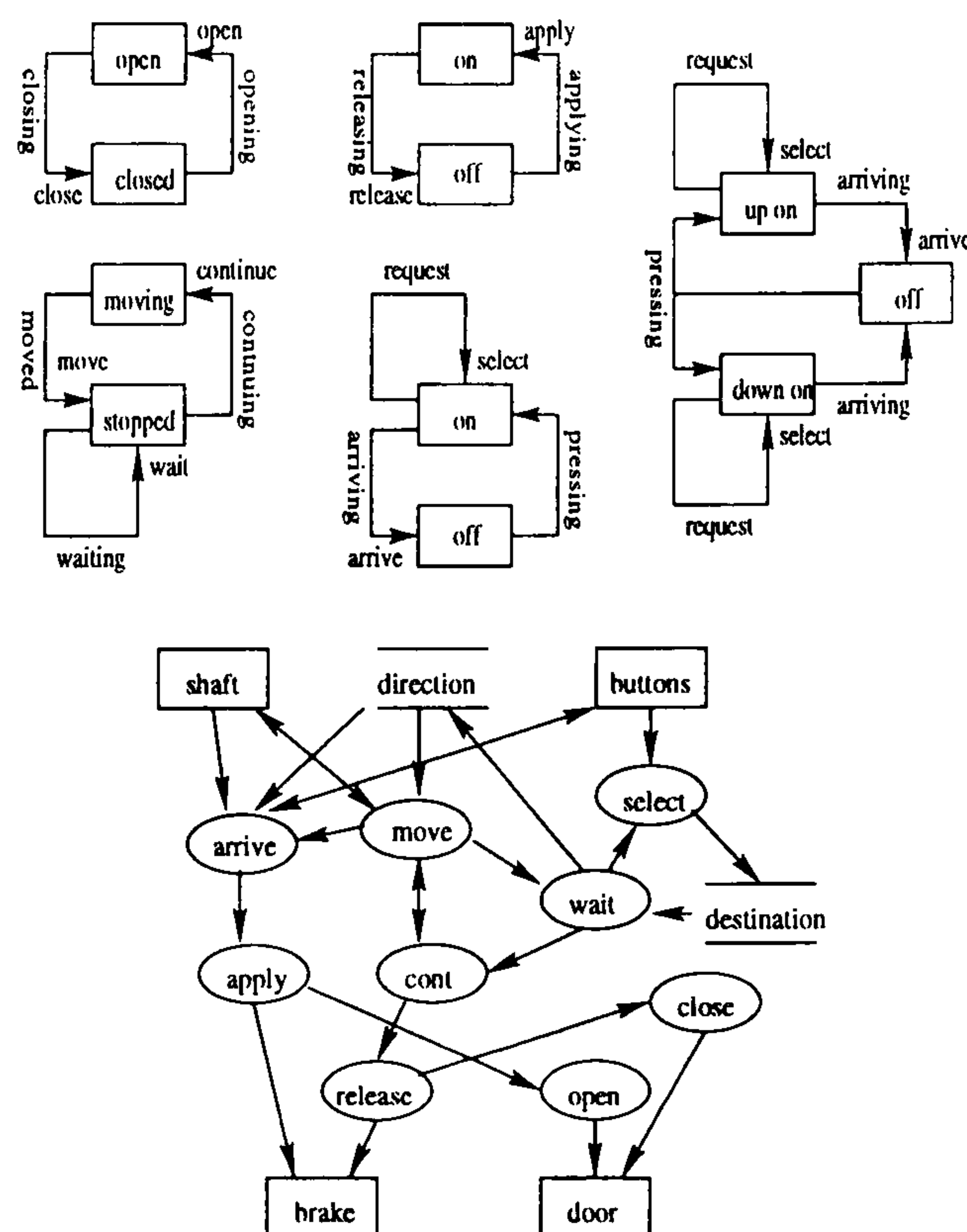


**Example 5.6. Association in SD.** The association between buttons, shaft, door and brake is shown in the MUL structure model



by the explicit representation of relations by directed arrows. The white arrow represents a structural relation and the black arrows represent functional relations between Object class instances. There is not necessarily any correspondence between the juxtaposing of Object class definitions and the juxtaposing of lift system components.

The functional relations represented in the structure model map onto relations between states and functions in the MUL behaviour and process models.



For example, the operate relations between the brake, shaft and buttons in the structure model correspond to the releasing and applying transitions in the behaviour model of the brake and the directed arrows feeding into the apply and release functions in the process model.

---

**Example 5.7. Synthesis in EM.** Synthesis of an LSD agent definition, such as the Hydrolift pump agent definition

```

agent pump() {
state
  change target
oracle
  brake pressure chan1
handle
  brake pressure chan2
derivate
  k = 100,
  change is (pressure < target) ? k :
            (pressure > target) ? -k : 0
protocol
  target == pressure + change && brake == OFF -> brake = ON,
  change == 0 -> target = chan1*k,
  pressure == target -> chan2 = target/k,
  brake == OFF -> pressure = pressure + change,
  brake == ON && change != 0 -> brake = OFF
}

```

involved identifying

- the observable features associated with the pump in the subject represented as state, oracle and handle declarations,
- the synchronization between observables associated with the pump in the subject represented as derivatives, and
- the causal relation between observables associated with the pump in the subject represented as protocol definitions.

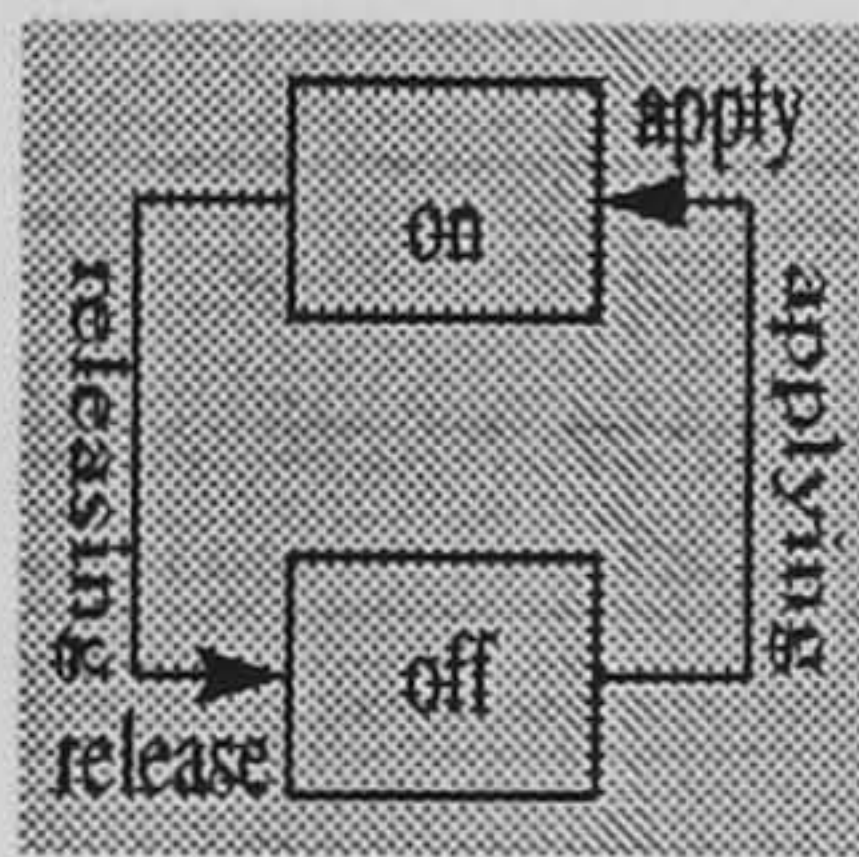
Construction of the visualization and animation was found, in general, to follow synthesis of the LSD specification when the subject was novel.

---



---

**Example 5.8. Synthesis in SD.** After synthesizing the behaviour model for the brake, as shown below, based on the software developer’s notion of its behaviour in terms of transitions between on and off states



he checked it against the requirements for the brake

... The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user ... The shaft mechanism releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

to verify the model was consistent with the rest of the MUL SD artefacts.

---



---

**Example 5.9. Transformation in SD.** The software developer followed a set method for transforming the requirements for the brake

... The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user ... The shaft mechanism releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.

into the artefact parts associated with the brake shown in Example 5.3. The resulting mappings were as follows:

- The nouns “brake” and “shaft” were transformed into object Class representations in the structure model.
- The verb phrase “the door is opened and closed by the brake” was transformed into a functional association between the brake and door Object classes in the structure model.
- The verb phrase “The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied” was transformed into a functional association between the brake and shaft Object classes in the structure model.
- The phrase “The brake is applied whenever the car arrives at a landing requested by a user” was transformed into a functional association between the brake and lift button Object classes in the structure model.
- The verbs “applied” and “released” were transformed into the actions and transitions of the brake Object class represented in the behaviour model.

Essentially, the mapping was from nouns to Object classes and from verbs and verb phrases to actions and functional associations.

---



---

**Example 5.10. Transformation in EM.** The transformation from the definition of the shaft LSD agent

```

agent shaft() {
  state
    floor destination direction
  oracle
    brake
  handle
    brake
  derivate
    direction is (floor < destination) ? UP :
                  (floor > destination) ? DOWN : NIL
  protocol
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}

```

to the definition of the shaft ADM entity

```

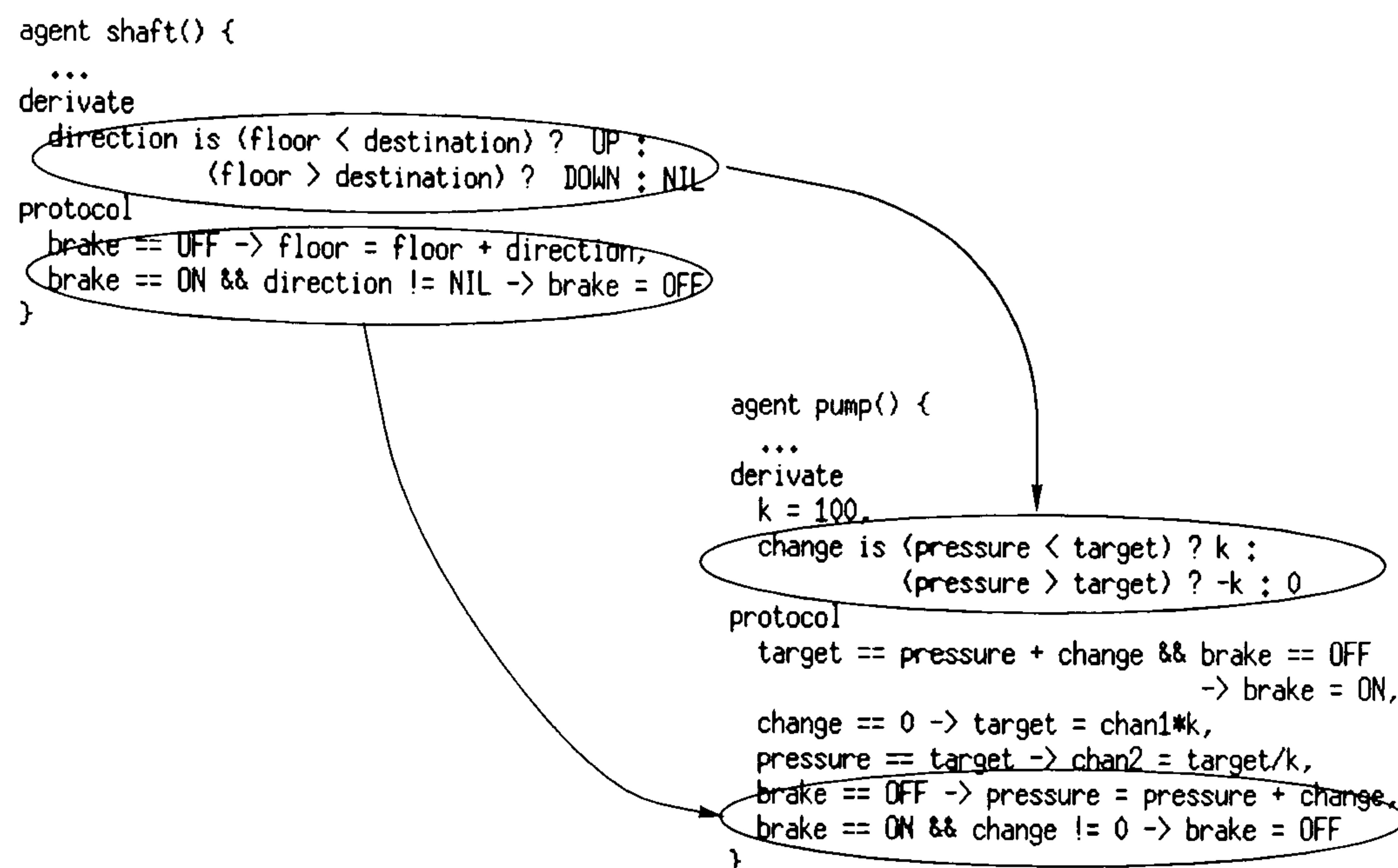
entity shaft() {
  definition
    direction is (floor < destination) ? UP :
                  (floor > destination) ? DOWN : NIL
  action
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}

```

is straightforward. However, although the surface-structure of the two definitions are very similar, their meaning is fundamentally different with the LSD representing a system in the world and the ADM definition representing a process in the computer. The difference between LSD and ADM is made clear in the introduction to EM in Chapter 2.

---

**Example 5.11. Transfer in EM.** The similarities between the structure of the LSD shaft and pump agent definitions is evidence of the analogical-like transfer between them in the lift project



The modeller kept the essential higher-level structure of the shaft definition while changing the observable names and adding some new definitions. In this way, the modeller preserved the notions of agency, causality and state that were represented in the LSD specification of the shaft agent.

In principle, a primitive LSD specification could have been generated to begin the EM railway project by changing the names of observables and agents in the MUL LSD specification:

landing	→	driver1
car	→	carriage
shaft	→	driver2
floor	→	station
landButton	→	schedStop
carButton	→	unschedStop
UP	→	NORTH
DOWN	→	SOUTH

This renaming transforms the MUL landing agent protocol into the protocol for a train driver

```

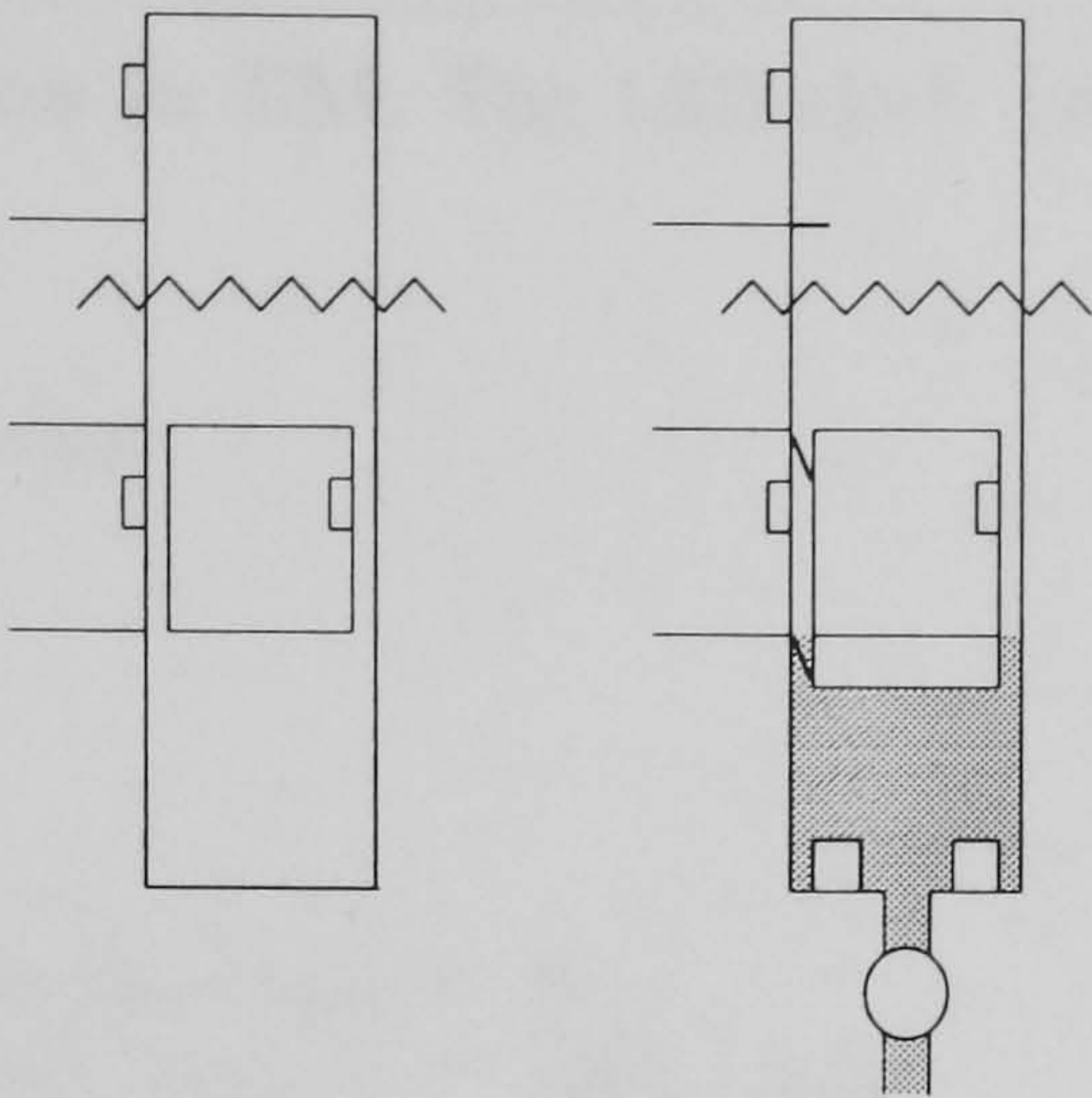
schedStop[_S] == NORTH && _S == station + 1 && brake == OFF -> brake = ON,
schedStop[_S] == SOUTH && _S == station - 1 && brake == OFF -> brake = ON,
schedStop[_S] != OFF && direction == NIL -> destination = _S,
station == _S -> schedStop[_S] = OFF

```

From this new protocol emerged novel timing characteristics. For example, whereas the observable `landButton` could change at any time the observable `schedStop` would only change during timetabling.

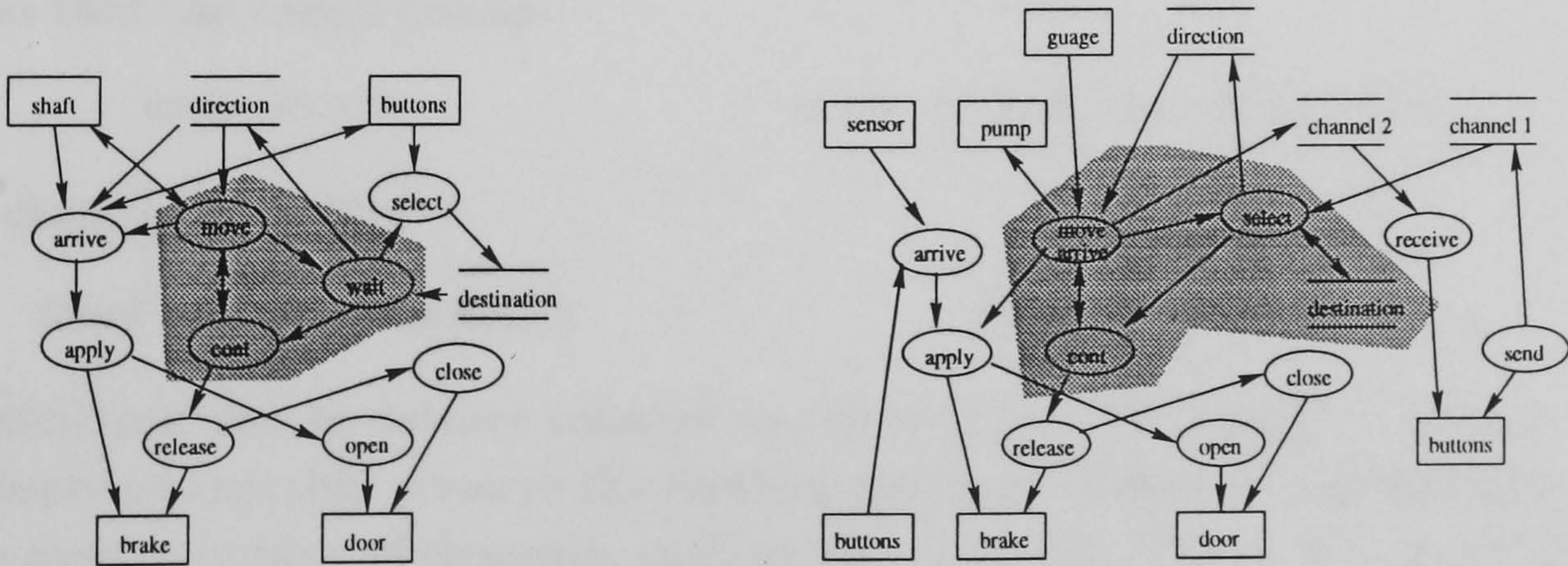


**Example 5.12. Transfer in PD.** The similarities between the sketches of the MUL and Hydrolift



is evidence of the analogical-like transfer that took place during the sketching of the Hydrolift. The designer kept the essential higher-level structure of the MUL sketch while adding new component representations.

**Example 5.13. Transfer in SD.** There are some quite fundamental differences between the structure of the process model of the shaft and the structure of the process model of the pump indicating that analogical-like transfer would have been of little use in generating the process model of the pump.



Shaft process model.

Pump process model.

The meaning of the process models are denoted more by its higher-level structure. Thus, a change in subject typically requires a major change in the structure of the process model.



---

**Example 5.14. Reduction in EM.** The LSD shaft agent definition

```

agent shaft() {
  state
    floor destination direction
  oracle
    brake
  handle
    brake
  derivate
    direction is (floor < destination) ? UP :
                (floor > destination) ? DOWN : NIL
  protocol
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}

```

can be reduced to a derivate and protocol definitions by removing the structurally higher-level agent construct

```

direction is (floor < destination) ? UP :
            (floor > destination) ? DOWN : NIL

brake == OFF -> floor = floor + direction,
brake == ON && direction != NIL -> brake = OFF

```

The protocol definition can be further reduced to statements representing the observations that cue causal change

```

brake == OFF                                brake == ON && direction != NIL

```

and the changes themselves

```

floor = floor + direction                    brake = OFF

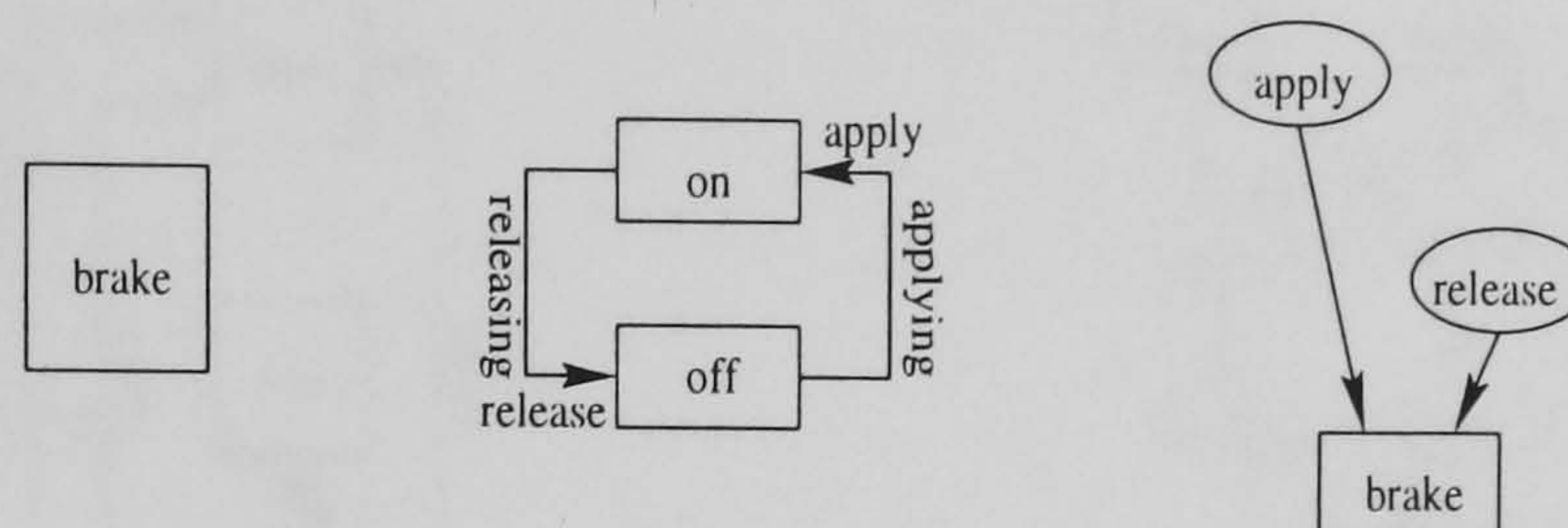
```

These definitions can be further reduced to observables. The products of each of these reductions arguably preserve the creative property of implicit meaningfulness. Even the most primitive of elements, such as the observable `floor`, has an implicit meaning. Once a definition has been reduced into basic elements those elements are typically reused in the synthesis of new LSD definitions.

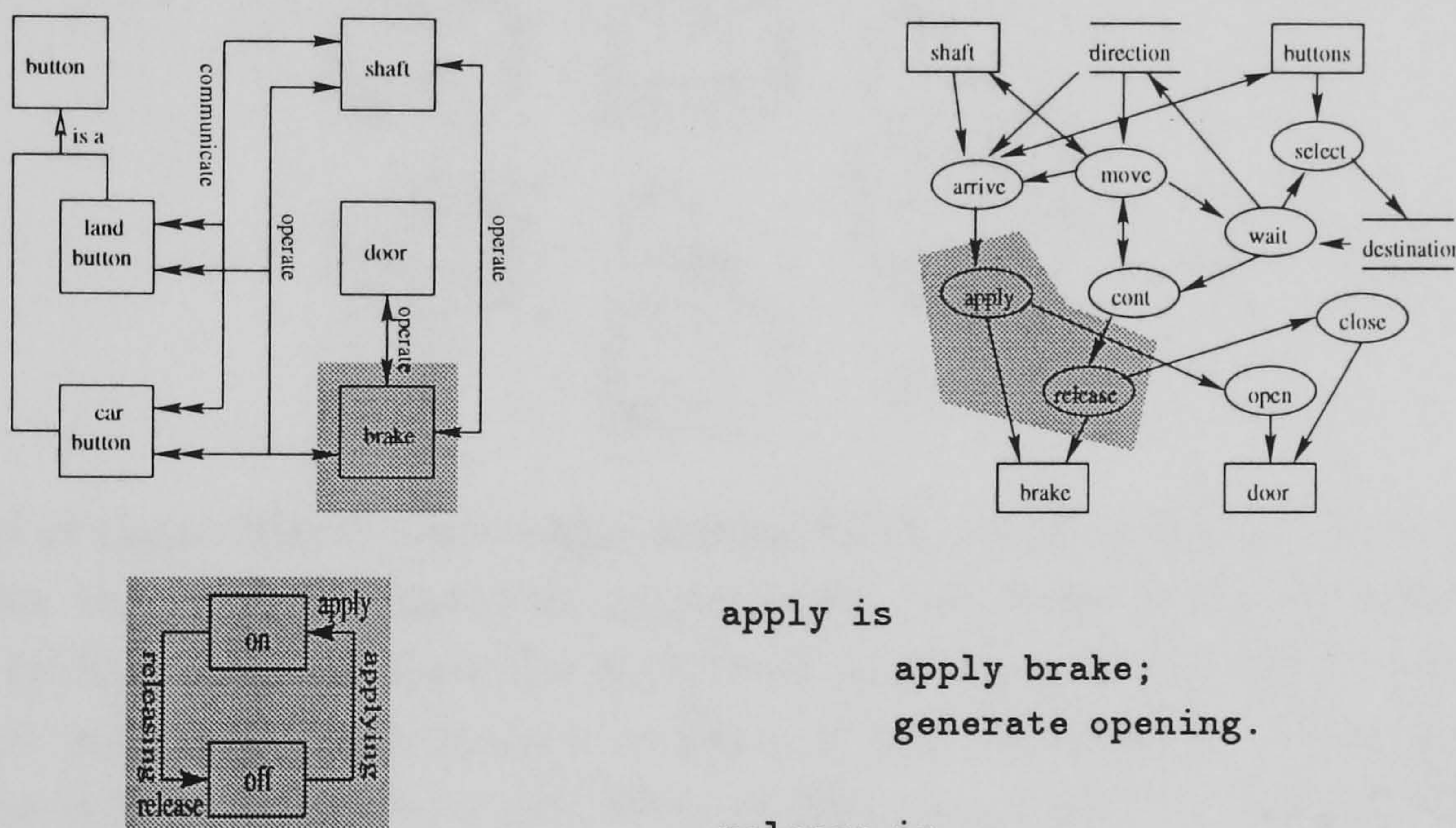
---



**Example 5.15. Reduction in SD.** It was found inappropriate to reduce structure, behaviour and process models down to basic elements, as shown below for the brake Object class, because most of the meaning of the models was denoted by their structure. So, reducing the structure of these models tended to “reduce” their meaning.



One solution was to keep the whole structure and highlight those parts associated with a particular Object class, as shown below for the MUL brake Object class.



apply is

```
apply brake;
generate opening.
```

release is

```
generate closing;
release brake
```

Another solution is was to define an interface that separated the models into the form of the Object class and its context. For example, the Eiffel-like [Mey88] definition of the door and brake Object classes

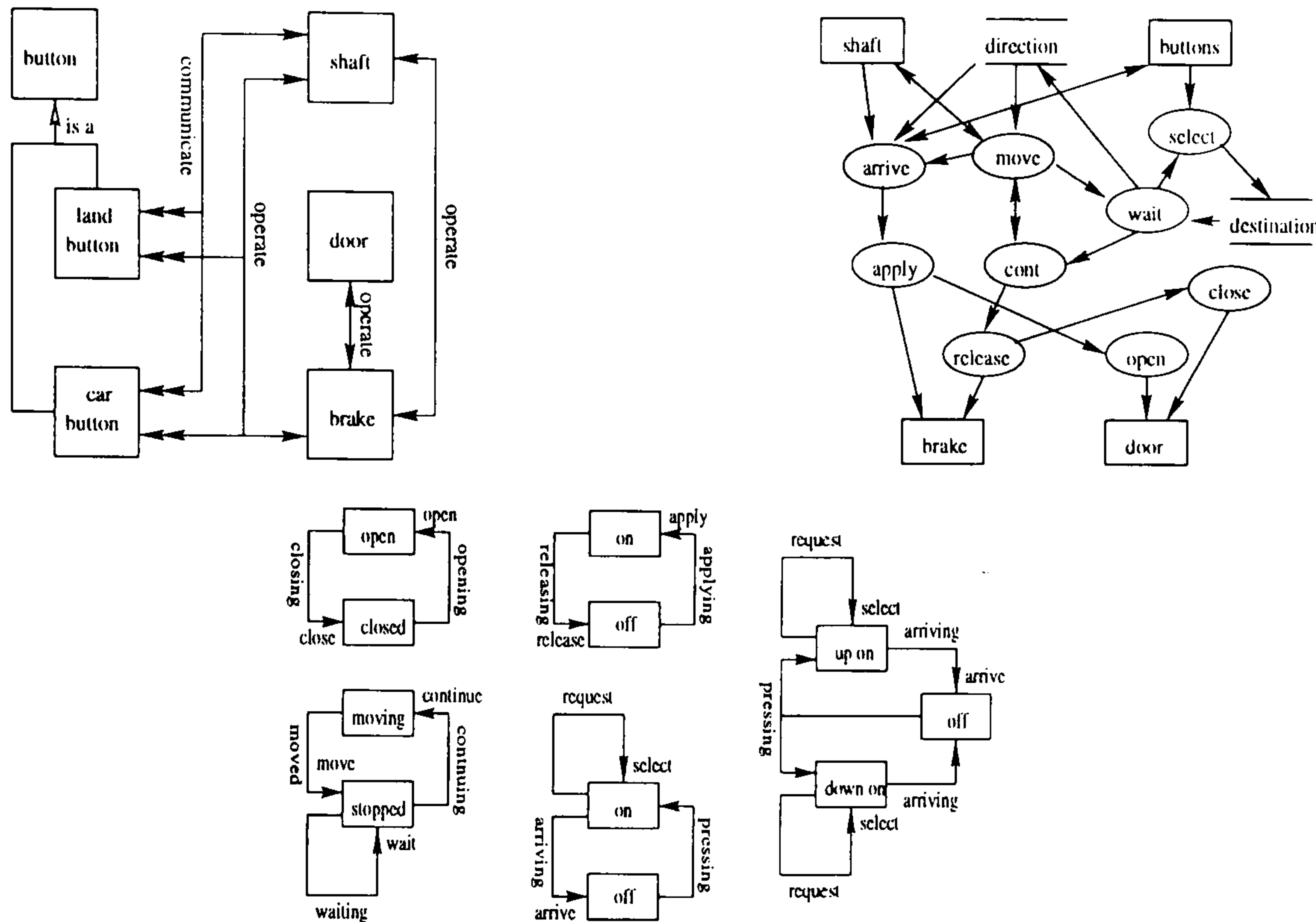
```
class DOOR open close
  open is {} body {status' = OPEN}
  close is {} body {status' = CLOSED}
  inv is {status = OPEN and brake.status = ON or status = CLOSED}
end

class BRAKE apply release
  apply is {} body {status' = ON}
  release is {} body {status' = OFF}
end
```

is typical of the way interfaces of Object classes are defined textually. Textual representation suits the abstract nature of the interface.



**Example 5.16. Lack of incentive for exploration in SD.** The MUL structure, behaviour and process models



are typical of the artefacts constructed during SD in the lift project. These artefacts (along with the MUL statement of requirements and action definitions not shown here but in Appendix C) have the analytical properties of familiarity, unambiguity, explicit meaning, completeness, consistency and convergence. These properties mean there is little incentive to creatively explore the artefacts. Take the definition of the brake as an example:

- the structure and function of the brake is defined explicitly;
- the symbolic language underlying the models of the brake is formal;
- the structural and functional meaning of the brake is unsituated;
- the SD method prescribes the generation of the brake model.

The structure, behaviour and process models of the brake, as well as the models of the other lift components, were generated in the lift project by the software developer transforming the statement of requirements into the artefacts of SD. The lack of incentive for exploration of the resulting artefacts meant that SD was an essentially generative activity.



---

**Example 5.17. Attribute finding in EM.** Searches around incongruous parts of artefacts typically results in the discovery of emergent features. One source of conflict within an EM model is the absence of oracle-handle pairs. An oracle-handle pair indicates a link between agents. In the MUL the oracles and handles of the car and shaft agents are

`floor direction brake destination`

The modelling of the Hydrolift involved defining the pump, sonar and sensor agents with the following oracles and handles

`pressure chan1 chan2 direction sensed`

Each set of observables belong to different domains. The MUL set are high-level concepts associated with use whereas the Hydrolift set are detailed concepts associated with engineering. The need to link the car, shaft, pump, sonar and sensor in the Hydrolift resulted in the creative exploration of alternative interpretations of the MUL observables:

- the floor was interpreted as the pressure of the column of liquid at the base of the shaft;
- the direction was interpreted as the signal from a direction sensor;
- the destination was interpreted as a target pressure.

These emergent features were attributed to the new agents, such as the pump agent

```
agent pump() {
state
  change target
oracle
  brake pressure chan1
handle
  brake pressure chan2
derivate
  k = 100,
  change is (pressure < target) ? k :
            (pressure > target) ? -k : 0
protocol
  target == pressure + change && brake == OFF -> brake = ON,
  change == 0 -> target = chan1*k,
  pressure == target -> chan2 = target/k,
  brake == OFF -> pressure = pressure + change,
  brake == ON && change != 0 -> brake = OFF
}
```

that shows the observables pressure and target pressure instead of floor and destination.

---

---

**Example 5.18. Conceptual interpretation in EM.** Modellers in the lift project interpreted the subjects in terms of the concepts of LSD, DoNaLD and ADM:

- LSD concepts correspond to common-sense notions, such as agency and causality, which help modellers describe unfamiliar objects and systems.
- DoNaLD concepts have formal meanings defining geometric shapes in a two-dimensional space that allow modellers to create visualizations on the computer and reason about the structure of objects and systems.
- ADM concepts have formal meanings defining processes that allow modellers to create animations on a computer and reason about the behaviour and functionality of objects and systems.

There are other languages in EM that help modellers conceptualize other aspects of the world that were not needed in the lift project, ARCA [Bey86a] for example. Providing a variety of languages, so that the world can be described in different ways and from different perspectives by the modeller, is an important principle in EM [BRS<sup>+</sup>89].

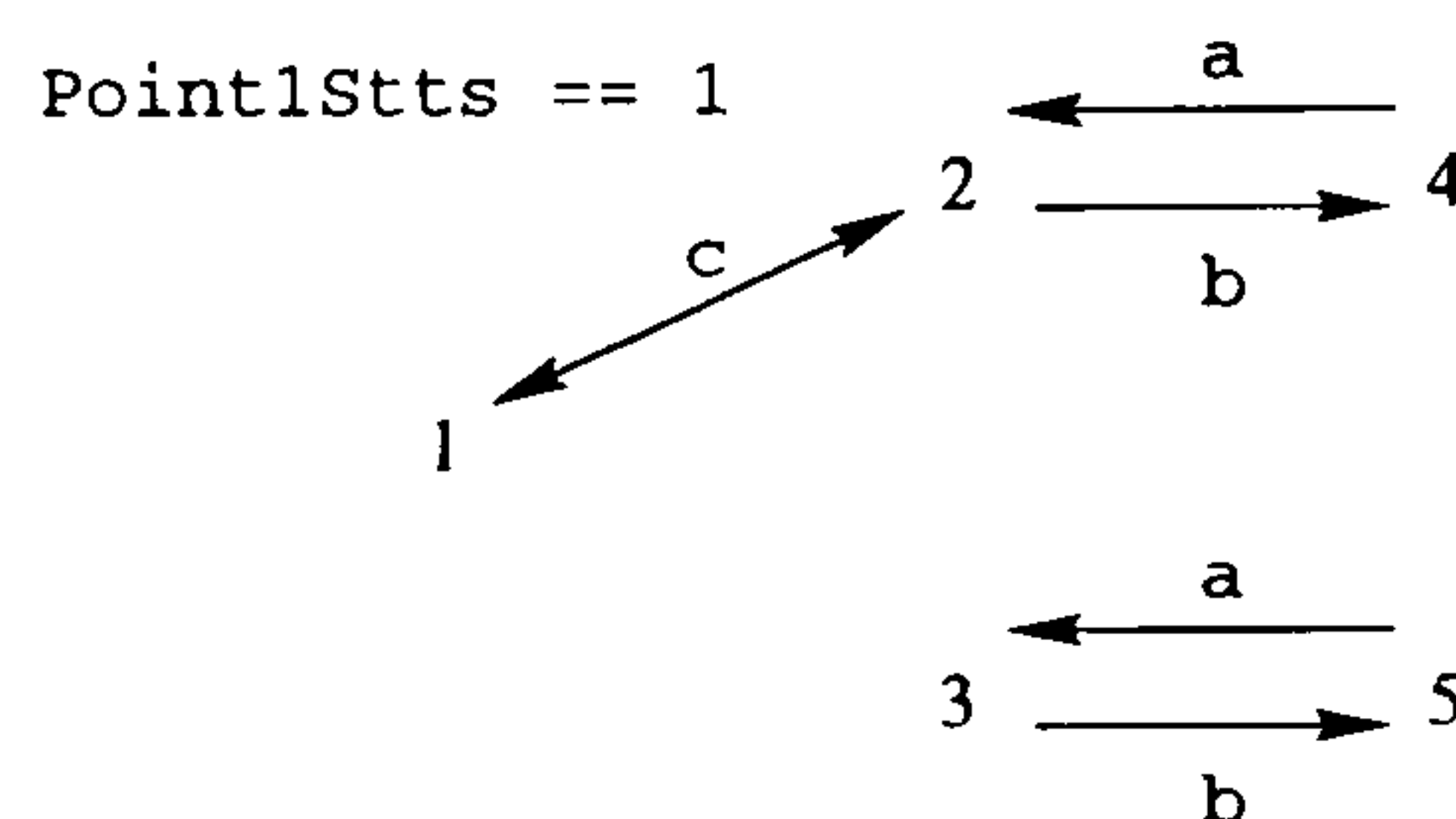
The railway project shows the use of conceptual interpretation to good effect. The connectivity of railway tracks was rather creatively interpreted as Cayley diagrams represented in the ARCA notation. For example, railway points are defined by the ARCA script

```

a_Point1{4} = 2
b_Point1{2} = 4
a_Point1{5} = 3
b_Point1{3} = 5
c_Point1{1} = if (Point1Stts == 1) 2 else 3
c_Point1{2} = if (Point1Stts == 1) 1 else 2
c_Point1{3} = if (Point1Stts == 1) 3 else 1

```

where Point1 is the diagram representing the railway point, the lower-case letters denote colours and the numbers denote vertices. The resulting diagram



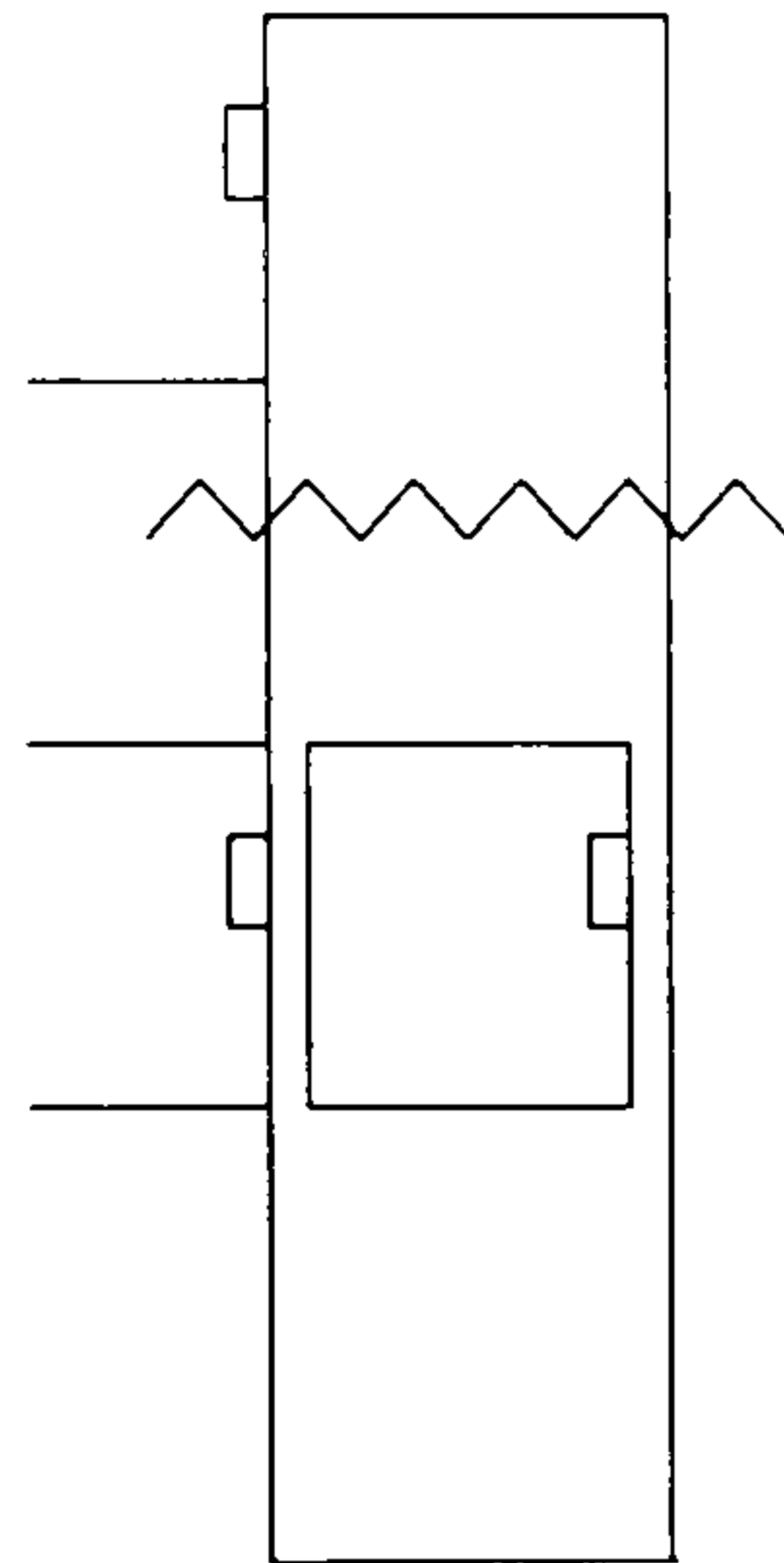
reflects the connectivity of a railway point that allows trains to switch between railway circuits in both directions.

---



**Example 5.19. Interpretation in PD.**

Although the sketch of the MUL, shown below, appears realistic the designer has clearly used conventions for representation, in other words, a kind of “graphical language” [Fer92]. The “vocabulary” and “grammar” of the language have to be known to fully understand the sketch.



Because lift systems have been around for so long designers have a standard repertoire of component “words” with which to interpret systems, as shown in Figure 5.2.

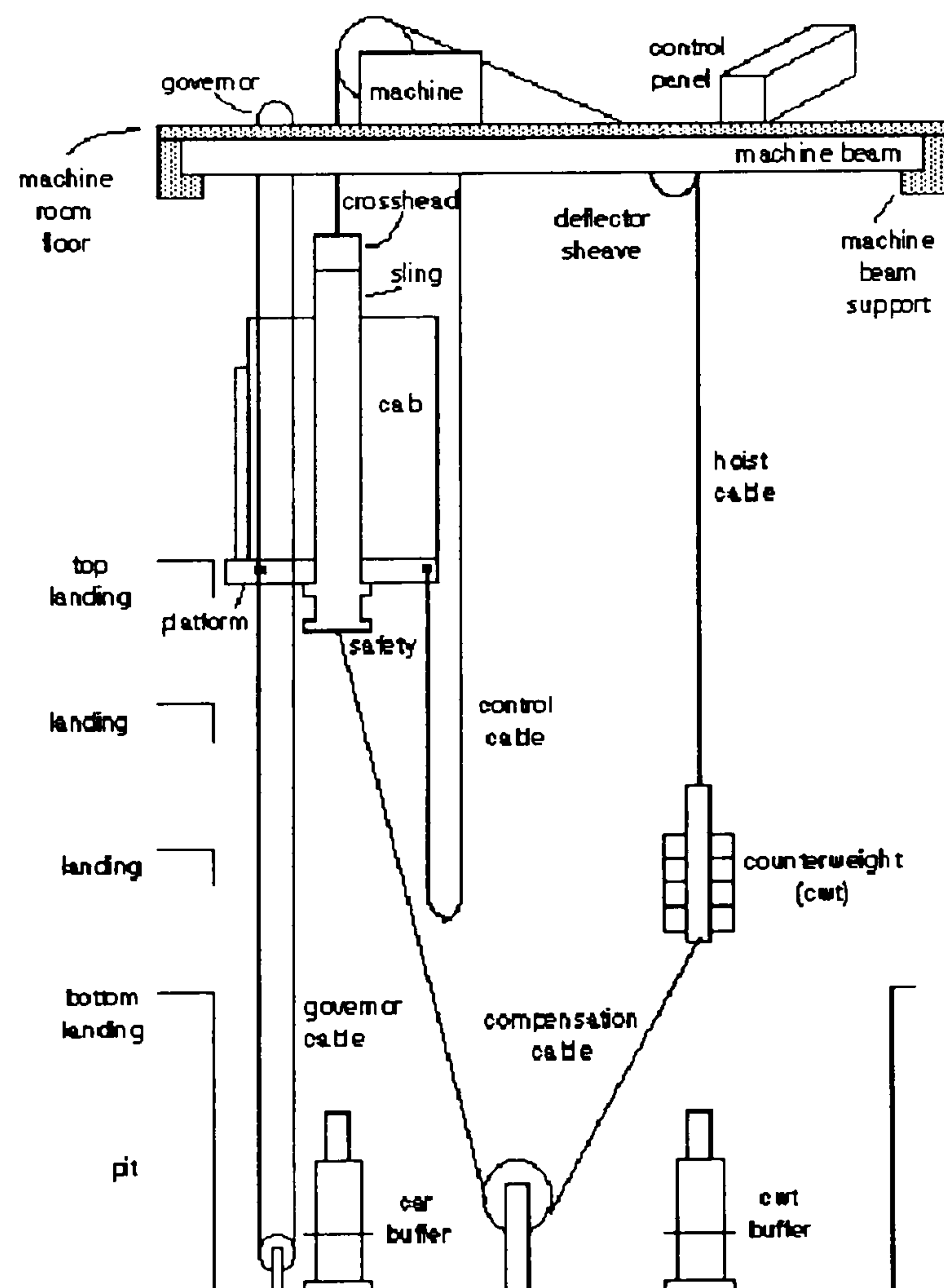
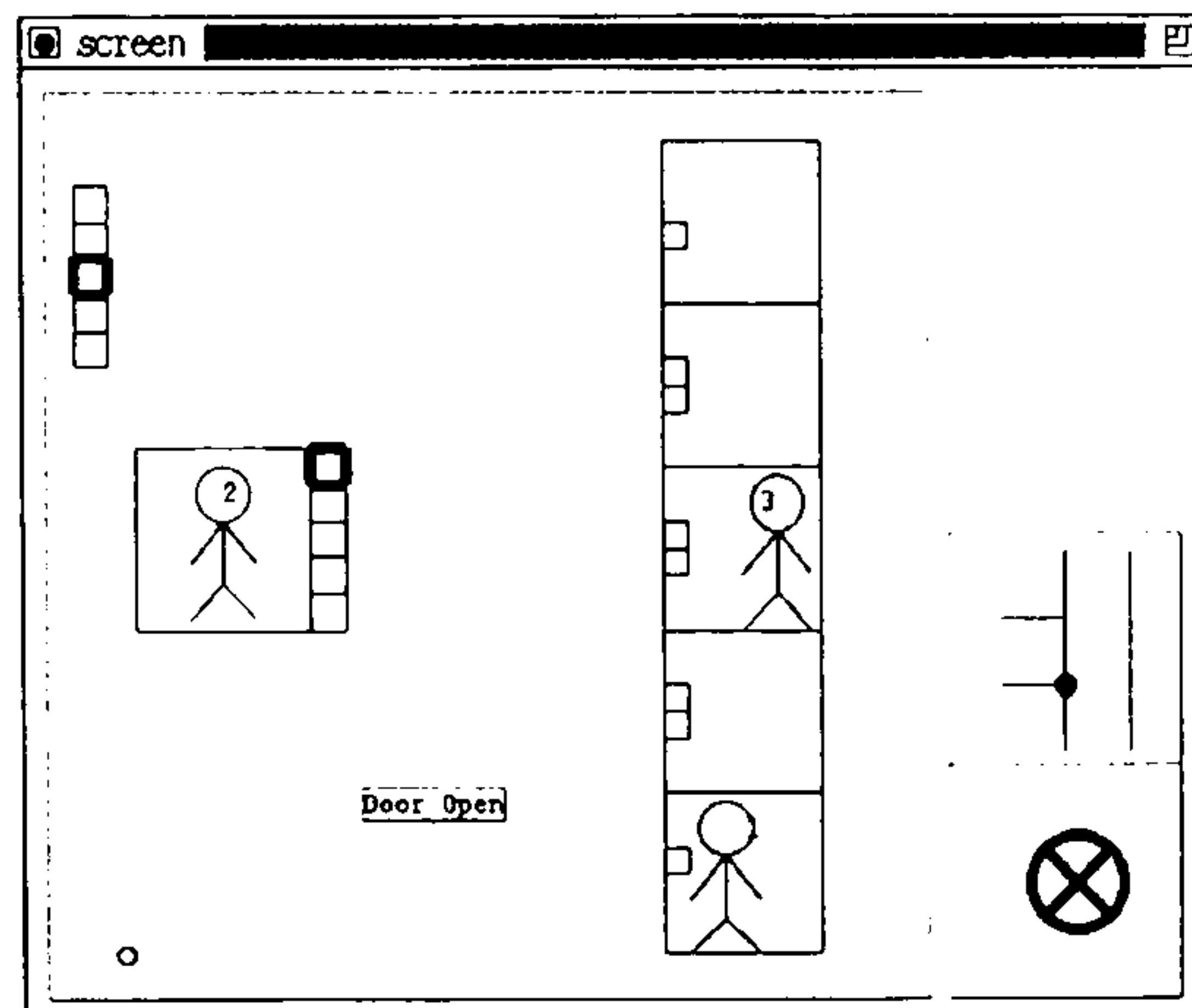


Figure 5.2: Lift system components.

This diagram is taken from [Yos92] which precisely defines each component in terms of its function and structure. In this way lift design becomes a process of fitting together components to give an intended function rather like constructing sentences is a process of fitting words together to give an intended meaning.

**Example 5.20. Functional inference in EM.** The screen-shot of the Hydrolift visualization and animation



shows a number of features that facilitated functional inference in the lift project:

- mouse point-and-click sensitive buttons corresponding to actual lift buttons;
- updated visual corresponding to the current state of the Hydrolift system;
- stick-men corresponding to lift users, such as the modeller.

These features of the visualization and animation, combined with the facility to enter redefinitions in a dialogue box, provided the modeller with an environment in which to infer the function of the Hydrolift.

By defining the LSD user agent, outlined below, the modeller was effectively modelling himself.

```
agent user(_U) {
  role In {
    state
      floor{_U}
    oracle
      door floor
    handle
      carButton
  }
  role Out {
    state
      floor{_U}
    oracle
      door floor
    handle
      landButton
  }
}
```

It was found that having a representation of oneself helped functional inference in the lift project and other EM projects, including the SBS (helmsman agent) and OXO (player agent).



---

**Example 5.21. Contextual shifting in EM.** Perhaps the most significant conceptual contextual shift in the lift project was when the LSD definitions for the pump, sonar and sensor

<code>agent pump() {</code>	<code>agent sonar() {</code>	<code>agent sensor()</code>
<code>  state</code>	<code>  oracle</code>	<code>  state</code>
<code>    change target</code>	<code>    chan1</code>	<code>    floor</code>
<code>  oracle</code>	<code>  handle</code>	<code>  oracle</code>
<code>    pressure chan1</code>	<code>    chan2</code>	<code>    direction</code>
<code>  handle</code>	<code>}</code>	<code>  handle</code>
<code>    pressure chan2</code>		<code>    sensed</code>
<code>}</code>		<code>}</code>

where added to the MUL LSD specification of a conventional lift system. This shift resulted in the emergence of details about the pump, sonar and sensor that were subsequently incorporated into the Hydrolift LSD specification, visualization and animation.

In the lift project the physical contextual shifts were limited to changing the person interacting with the visualization or animation. This resulted in an objective evaluation of the Hydrolift design:

- the evaluation of the Hydrolift from multiple perspectives of various designers;
- the evaluation of the Hydrolift from multiple perspectives of various users.

Ideally the interaction between computer and its environment would not be restricted to human interaction. In the future it is hoped that the computer can play a more situated role by interacting with actual components.

The OXO project shows the use of conceptual contextual shifting in EM to good effect. Two observations of the game of OXO were modelled:

- the observation of the rules based on the conventional 3-by-3 matrix;
- the interpretation of the board in terms of the conventional 3-by-3 matrix.

The consequence of this approach was that the model of the rules was generic for all games of OXO whilst the model of the interpretation of the board changed for different geometries of the board. Different scripts represented the different player interpretations:

- a conventional board (`geomoxo.e` and `oxo.geom`);
- a three-dimensional board (`geomoxo4.e` and `oxo4.geom`);
- a board over a classical finite projective plane (`geompp7.e` and `pp7.geom`).

Each of these geometries corresponded to a different context for the player.

---

## Chapter 6

# SD as Systems Development

Chapters 3, 4 and 5 have shown that the artefacts and actions of EM and PD are different from those of SD. This suggests that EM and PD cannot be used as an approach to developing software in the conventional sense. EM and PD are more appropriately applied to the creative development of innovative systems than the methodical transformation of requirements into software that characterizes SD. One way that EM and PD could be construed as approaches to developing software is if SD could be viewed as systems development.

This chapter considers how EM and PD might be used as an approach to developing software based on a generalization of the notions of computer, program and programming. The conventional view of the computer as an electronic device, or embodiment of a Turing machine, is generalized to *computer as artefact*, program as stored program is generalized to *program as system configuration*, and programming as SD is generalized to *programming as configuring systems* that is essentially the activity of EM and PD. The usefulness of this alternative view is assessed by considering how it addresses the topical issues in SD and requirements engineering.

### 6.1 Generalizing computers, programs and programming

The concepts of computer, program and programming have precise and unambiguous meaning in SD. The notion of computer in SD is characterized by the electronic computer, based on the von Neumann architecture [Asp90, EE90], and Turing's



model of a universal machine [Tur36, SW88], consisting of an unbounded store and finite control unit capable of performing simple operations. Associated with this notion of computer is the concept of the stored program [Asp90, EE90]: a sequence of symbols that are the same as data but can be interpreted as actions by the computer. Subject to this interpretation, programming is the process of constructing the sequence of actions that is the stored program. Constructing the stored program is typically facilitated by the generation of SD artefacts, such as the structure, behaviour and process models and code written in programming languages.

In Chapter 2 the EM notion of the computer as artefact was introduced. The meaning of the term artefact, as used in this thesis, characterized by the abstract boundary that separates the world into form and context, was introduced at the beginning of Chapter 4. The idea of the *computer as artefact* offers a more general concept of the computer than in SD with the electronic computer and Turing machine being particular computer forms. The goal of both electronic computers and Turing machines - to perform the sequence of actions in store - can be realized by other forms given a suitable interpretation of the terms program and programming. This section considers the appropriateness of interpreting programs as system configurations that store the actions of the system within the arrangement of components. Subject to this interpretation, programming is the process of configuring a computer artefact that typically involves activities like EM and PD.

### 6.1.1 Computer as artefact

In SD the electronic computer and Turing machine embody the notion of the computer. In the lift project the software developer constructed the structure, behaviour and process models during analysis with the intention of using them to design and implement code to execute on a electronic computer. The electronic computer largely determines the nature of the models, methods and tools of SD.

It is an empirical fact that almost all electronic computers consist of the same essential elements [NS76]:

- the store containing sequences of actions and associated data (programs);
- the processor capable of performing the actions in store;

- the input and output devices that support interaction between the processor and environment.

This architecture corresponds to the Turing machine model that contains the essentials of all computers, in terms of what they can do, though other computers with different memories and operations might carry out the same computations with different requirements of space and time. In particular, the model of a Turing machine contains within it the notions both of what cannot be computed and of universal machines - computers that can do anything that can be done by any other machine [Tur36, SW88].

In EM and PD the electronic computer and its environment are combined within the notion of the computer artefact. The meaning of the term artefact as used in this thesis is given at the beginning of Chapter 4. Central to this is Simon's characterization of artefact in terms of form (inner-environment), context (outer-environment) and purpose (the terms form and context are used by Alexander in a similar characterization [Ale67]):

An artifact can be thought of as a meeting point - an "interface" in today's terms - between an "inner" environment, the substance and organization of the artifact itself, and an "outer" environment, the surroundings in which it operates. If the inner environment is appropriate to the outer environment, or vice versa, the artifact will serve its intended purpose [Sim81].

Adopting this view, the electronic computer and Turing machine are particular forms of the computer artefact. The form is only half of the computer artefact. The computer artefact includes the form, such as the electronic computer or Turing machine, and its context.

The computer artefact is characterized by the goal shared by all computers, including the electronic computer and Turing machine, to perform the sequence of actions in store. The early electronic computers satisfied this goal in virtual isolation. However, an increasing number of actions performed by electronic computers today require corresponding actions in the environment. In combination the store,



processor and input/output devices provide the physical means for satisfying the goal but the successful operation of the electronic computer depends on appropriate actions in its environment: “the form of an artefact is a collection of natural phenomena capable of attaining the goal in some range of environments and the context determines the conditions for goal attainment” [Sim81].

Whereas SD has traditionally placed emphasis on the form of the electronic computer alone, the computer artefact gives equal importance to form and context. Perhaps the main reason for this emphasis in SD is that few generalizations can be made about the context of electronic computers because of their many and varied applications. However, one can say that in general the context of electronic computers typically consist of two kinds of elements:

- people and
- mechanical (including electromechanical) devices.

In the lift project it was observed that the people surrounding the computer at any time tended to be organized into communities, such as modellers, product designers, software developers, lift users and lift engineers. Communities used the computer for a variety of purposes, for example, the modellers and designers used it for creative modelling and design whereas the software developers used it for analysis. It is clear from the artefacts discussed in Chapter 4 that the computer controller would be situated among mechanical components - doors, motors, pumps, buttons, levers - in the lift system.

Modellers and designers use the notion of the computer as artefact to consider alternative computer forms, or to ignore the computer form altogether, during EM and PD. Simon identifies the two related principles of predictability and generality that result from the characterization of artefact in terms of form and context [Sim81]:

- it is often possible to predict behaviour from knowledge of the goal and context, with only minimal assumptions about the form;
- often quite different forms are capable of accomplishing identical or similar goals in identical or similar contexts.

These principles are utilized in EM: predictability means that modellers can focus attention on modelling the context to predict the behaviour of the computer form; generality means that the modeller can consider alternative computer forms so long as they satisfy the same goal as electronic computers and Turing machines in a given context.

The EM notion of the computer as artefact was first introduced in Chapter 2 when describing the unusual status given to the `tkeden` interpreter. In EM the computer is only significant in so far as it serves as a physical instrument with which the modeller interacts. This is in contrast to the way in which the computer is conventionally regarded in classical computer science as a means of implementing an abstract algorithm or computation. In effect, it is how the user apprehends the computer as a physical object that matters in EM, not the invisible mechanism by which the object is specified [BNR95]. As explained in Chapter 2, the view of computer as artefact is necessary for the 1-agent approach to modelling that is essential to EM.

### 6.1.2 Program as configuration

Conventionally the program is the sequence of actions stored in an electronic computer or Turing machine. This is reflected in the dictionary definition of the term program: “the sequence of actions to be performed by an electronic computer in dealing with data of a certain kind” (cited in [BR92]). The notion of program and program execution has a precise and unambiguous meaning with respect to the von Neumann architecture of the digital computer, consisting of a store, processor and input/output devices, and abstractions thereof [Asp90]. This section considers the meaning of the term program when “computer artefact” is substituted for “electronic computer” in the above dictionary definition [BR92].

When the computer form is an electronic computer or Turing machine the goal of the computer artefact - to perform the sequence of actions in store - embodies the stored program principle. When the form is an electronic computer the meaning of the terms action and store in the goal are precise and unambiguous:

- the actions of the processor are simple read, write and logical operations;



- the store is a large number of neighbouring locations each holding a binary digit denoting an action or data.

Correspondingly, when the computer form is a Turing machine the actions of the finite control unit are specified in terms of simple operations - read, write and scan operations - on the store. The store is typically construed as an infinite tape divided into squares that each hold symbols from a finite alphabet each denoting an action or data [Tur36, SW88].

It has been shown in previous chapters that the structural and functional aspects of a stored program are represented by the artefacts constructed during SD. The structure of the artefacts typically reflect the structure of the stored program: symbols representing actions organized into sequences. The meaning of the artefacts is given, for the most part, by the semantic relationship between the symbols in the artefacts and actions of the processor in a digital computer. Such artefacts were constructed and used during SD in the lift project:

- C++ programs;
- structure, behaviour and process models;
- structure of the statements of requirements.

Each kind of artefact was associated with a particular purpose in the lift project:

- the principal purpose of the C++ program was to be automatically translated by the electronic computer into a stored program;
- the purpose of the structure, behaviour and process models was to assist the software developer in analyzing the requirements of the lift system and represent the information necessary to design and implement a C++ program;
- the software developer used the logical structure of the statements of requirements to construct the structure, behaviour and process models.

All the above representations were linked together into a closed system by automatic translation and manual methods of analysis.

By considering alternative computer forms, such as a mechanical device, a broader view of the stored program emerges. When the computer form is a mechanical system the goal of the computer artefact - to perform the sequence of actions in store - requires a broader and more common sense interpretation of the terms action and store than are associated with electronic computers and the Turing machine model:

- the actions of the mechanical device are those that are made possible by the configuration of components, such as open, close, set, reset, raise, lower, fill and empty;
- the actions are stored within the configuration of components in the mechanical device.

Based on this interpretation the program can be thought of as the configuration of a system. Designers learn the relationship between the form and structure of mechanical devices through the experience of developing systems.

By virtue of the symmetrical nature of interactions, the structure encodes the *sequence* of actions performed by the system in the same way that the stored program describes the sequence of actions performed by a digital computer. Interaction, by definition, involves the synchronization of two actions (this view is adopted in the process calculi CCS [Mil89] and CSP [Hoa85]). An interaction between the system and its context involves an action of the system synchronizing with an action in the context - there is an essential symmetry between form and context. For example, the lift door opening synchronizes with a user pressing a button. By adopting this view of interaction it follows that it would indeed be possible, in principle, to determine the sequence of actions of a system by decoding its structure. The timing of the actions performed by the system may depend on the context but for every action in the context the system is ready with its counterpart action.

It has been shown in previous chapters that the configuration of mechanical systems, like the lift system, can be represented by the artefacts constructed during EM and PD. The structure of the artefacts typically reflect the system structure: metaphorical representations of components organized in space. The meaning of the



metaphors are implied by their shape and context. Such artefacts were constructed and used during EM and PD in the lift project:

- design sketch;
- LSD specification;
- visualization and animation.

Each kind of artefact was associated with a particular purpose in the lift project:

- the purpose of the sketch was to help the designer create a satisfactory design for the system;
- the purpose of the LSD specification was to help the modeller conceive the system in terms of observables and agency without having to think abstractly in terms of structure and function;
- the purpose of the visualization and animation was to creatively explore the structural and functional features that emerge from analysis of the LSD specification.

The artefacts constructed during the lift project were related by the fact that each tended to be progressively more detailed and more abstract.

There are clearly similarities between the nature of SD artefacts and the artefacts constructed during the later stages of EM and PD. In PD the detailed descriptions of components and drawings showing how components are to be arranged are more formal than the sketches produced during conceptual design. Similarly, the scripts defining the animation in the later stages of EM define the function of a system more precisely than the LSD specification or visualization. The general characteristics of the artefacts constructed later in EM and PD have been identified in previous chapters:

- the artefacts are less creative and more analytical than the earlier artefacts;
- linguistic patternment is more important in the artefacts than in the earlier artefacts;

- the meaning of the artefacts is less situated than the meaning of earlier artefacts.

In this way, the SD artefacts in the lift project can be thought of as being similar in nature to the artefacts constructed later in EM and PD that are more analytical, linguistic and unsituated than the artefacts constructed earlier in EM and PD.

The view of a program as a system configuration addresses the essential difficulties of software - complexity, conformity, changeability and invisibility - as identified in Brooks' influential paper entitled "No Silver Bullet: Essence and Accidents of Software Engineering" [Bro87]:

- the complexity of mechanical systems is easier to comprehend than software;
- interfaces between mechanical systems are less of an issue than between software elements;
- modification is essential to mechanical systems but seen as a problem in software;
- mechanical systems are visualizable whereas software is essentially invisible.

A more in-depth discussion of how the concept of the program as configuration addresses the essential difficulties of software follows in Section 6.2.

### 6.1.3 Programming as configuring

The conventional notion of programming is associated with the construction of a program stored in an electronic computer. In SD the construction of a stored program is given a precise and unambiguous meaning by the methods and models used by the software developer. This section considers the meaning of the term programming when "computer artefact" is substituted for "electronic computer" and "stored program" is substituted for "system configuration" above.

Programming corresponds to SD when the program is stored in an electronic computer. Programming is traditionally used to describe the activity of coding within SD. However, since the purpose of the preceding stages of SD is to determine what is to be coded, the whole process can be construed as programming. In fact,



the SD methods were motivated by the need to elevate programming from a craft to an engineering discipline [Gib94].

Chapter 2 introduces SD as essentially the process whereby a statement of requirements is transformed into code. Most SD methods share the same sequence of stages as were followed during SD in the lift project:

- analysis of the statement of requirements resulting in the construction of structure, behaviour and process models;
- design of the software components and architectures based on the models constructed during analysis;
- coding of the design elements in a suitable programming language.

Each of these stages is characterized by a method to be followed by the software developer and the artefacts that are constructed by following the method. In combination these methods and artefacts are a system animated through the agency of the software developer. This formal system is linked to the stored program by the compiler or interpreter that automatically translates the code into binary digits stored within the electronic computer.

By considering alternative computer forms, such as a mechanical device, a broader view of programming emerges corresponding to EM and PD. When the program is a system configuration programming corresponds to configuring the system. Configuring a system is traditionally associated with manufacture in PD when components are arranged and connected together. However, since the purpose of the preceding stages of PD is to determine the components, arrangements and connections, the whole process can be construed as configuring the system. The stages that precede manufacture are what make configuring the system an engineering discipline instead of a craft [Fer92]. In this way, EM can also be thought of as configuring systems, although there is no product, because it corresponds to the stages that precede manufacture in PD.

Chapters 4 and 5 have identified similarities between the artefacts and techniques in EM and the stage of conceptual design in PD. The PD stage of conceptual design and the related stage of detail design were introduced in Chapter 2:

- conceptual design involves the creative generation and evaluation of design concepts;
- detail design involves focusing on the details of the design concept determining which subsystems and components will provide the desired functionality.

The conceptual design phase is characterized by the creative use of sketches. The detail design phase is characterized by the use of previous knowledge about components, represented precisely and unambiguously as labeled drawings, tables, specifications, formulae and the like, in order to inform the realization of the design concept. The system is manufactured using the detailed drawings produced at the end of detail design.

This progression in conceptual design from the simple to the more detailed concept of a system corresponds to the process of conceptualization that characterizes EM [Bey97] described in Chapter 2:

- interaction with artefacts: identification of persistent features and contexts;
- practical knowledge: correlation between artefacts, acquisition of skills;
- identification of dependencies and postulation of independent agency;
- identification of generic patterns of interaction and stimulus-response mechanisms;
- non-verbal communication through interaction with similar environment;
- situated use of language;
- identification of common experience and objective knowledge;
- symbolic representation and formal languages: public conventions for interpretation.

These stages represent a progression from a subjective to an objective view of the subject. The early stages correspond to the modeller's view of the subject during 1-agent modelling. In the later stages the modeller develops methods of communication as typified in n-agent modelling. Finally, the model acquires a meaning independent of the subject and modeller (0-agent system).



There are similarities between the process of SD and the later stages of EM and PD. The later stages of PD involve an engineer using detailed descriptions of components to determine what arrangements will provide the desired functionality. Similarly, the later stages of EM involve the modeller writing sections of definitive script in order to generate the desired behaviour. The general characteristics of the later stages of EM and PD have been identified in previous chapters:

- the later stages involve less creative exploration of artefacts and the subject than earlier stages;
- actions in the later stages are more predictable than actions in the earlier stages;
- the later stages follow general techniques unlike the earlier stages that are determined more by the specific situation.

In this way, SD can be thought of as corresponding to the later stages of EM and PD when the subject has been conceived and represented formally.

The earlier stages of EM and PD arguably provide a more appropriate framework for requirements engineering than the traditional view of requirements engineering as an extension to analysis in SD. The traditional view of requirements engineering is of the process that generates the statement of requirements previous to SD [Poh96, Dav93, Hof93]. The need to integrate these two activities has resulted in established SD techniques, such as object-oriented techniques, being adopted in requirements engineering. In contrast, the view of requirements engineering as EM and PD is of a continuous process of conceiving a system that progresses in parallel with evolving artefacts. The similarities between EM and PD and the predictions for new directions in requirements engineering identified by Siddiqi and Shekaran, in particular the importance of context, are discussed in Section 6.5.

The activities of EM and PD are in the same spirit as the ways to attack the essential difficulties of software identified by Brooks in [Bro87]: buy versus build; requirements refinement and rapid prototyping; incremental development; great designers.

- reusing artefacts instead of constructing new ones from scratch reduces the cost of invention and at the same time gives continuity;
- exploring the system as it is conceived in EM allows users to clarify their requirements whilst interacting with an up-to-date prototype;
- conceptualization of a system is essentially an iterative and incremental process;
- good conceptual design requires skilled designers.

A more in-depth discussion of the correspondence between the view of programming as EM and PD and the attacks on the essential difficulties of software follows in Section 6.3.

## 6.2 Addressing the essential difficulties of software

In his acclaimed paper entitled “No Silver Bullet: Essence and Accidents of Software Engineering” [Bro87] Brooks identifies four essential properties of software - complexity, conformity, changeability and invisibility - that are the root cause of problems in SD. In his recent anniversary edition of “The Mythical Man-Month” [Bro95] Brooks develops his theme of the four essential properties of software - complexity, conformity, changeability and invisibility - by reacting, in particular, to the rebuttal paper by Harel entitled “Biting the Silver Bullet” [Har92]. This section considers how these properties are addressed by the view of a program as system configuration in EM and PD and the associated views of the computer as artefact and programming as configuring systems.

### 6.2.1 Complexity

Brooks makes the observation that software entities are more complex for their size than perhaps any other human constructs because no two parts are alike (at least above statement level) (Table 6.1). A central principle in SD is that similar objects are represented abstractly by a single class. However, as Brooks points out, in this respect, software differs profoundly from computers, buildings and other



Software ...	Systems ...
has no two parts that are alike	have repeated parts
typically has more states than systems	typically have fewer states than software
complexity increases exponentially	complexity increases linearly
complexity is rampant	complexity is managed

Table 6.1: Contrasting complexity in software and systems.

systems, where repeated elements abound. In EM and PD the modeller and designer tend to represent each instance of a part within a system in order to keep a close correspondence between the subject and its representation.

EM has a principled approach to reducing the number of repeated parts in an LSD specification that is not based on classification as in SD. The approach is directly related to the two complementary principles that constitute concurrent engineering in EM [ABCY94c, ABCY94a, ABCY94b] introduced in Chapter 2:

- specifying agents in LSD by considering them in isolation;
- introducing a context for interaction by animating agents using ADM.

It is the context of agents with the same set of observables, dependencies and protocols that distinguish them from one another. For example, it is the position of a person in a lift system that distinguishes them from other people and the position of a brick in a wall that distinguishes it from other bricks. Consequently a single LSD agent specification can represent similar agents by considering them in isolation. The context of agents is the focus of visualization and animation in EM.

Brooks argues that one of the major problems that faces software developers in SD is in dealing with the enormous number of states that a piece of software can have. This is especially true of concurrent programs in which subprograms are changing states independently. Large numbers of states makes it difficult for software developers to conceive, describe and test software. In EM and PD the modeller and designer is not concerned with program states but with the observed states of the subject [BRY90]. Brooks recognizes that software has orders-of-magnitude more states than computers do. In EM and PD the modeller and designer focuses on



the mechanical and electro-mechanical devices that tend to have fewer states than electronic computers or software.

As was discussed in Chapter 4, all relationships between software parts have to be represented explicitly in SD because of its abstract nature. In contrast, relationships that are implied in the subject are not explicitly represented in the model or sketch in EM and PD due to the direct correspondence between the subject and its representation. Brooks observes that, the software elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly. In EM and PD the complexity of structural and functional relationships are dealt with by the powerful computational framework provided by the ADM and the powerful mental processes of the designer familiar with interaction between components.

Brooks concludes his section on complexity by stating his belief that the complexity of software is an essential property: “descriptions of a software entity that abstract away its complexity often abstract away its essence” [Bro87]. Complexity is also important to the product designer who cannot guarantee the quality or safety of an innovative product based on generalized models alone [Pug91, Pug96, Fer92]. The designer must embrace the unfathomable complexities of nature. Similarly, the complexities of natural phenomena are taken head-on by the modeller in EM.

### 6.2.2 Conformity

Brooks argues that, although scientists have to face complexity, they have a firm faith that there are unifying principles to be found in nature (Table 6.2). He points out that Einstein argued there must be simple explanations of nature because God is not capricious or arbitrary.

Brooks believes that no such faith of unification and simplicity comforts the software developer. Much of the complexity that he must master during SD is arbitrary, forced by the many human institutions and systems to which his interfaces must conform. He argues that these differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.



Software ...	Systems ...
has no principle of unification	has the natural principle of unification
conforms to complex interfaces	have few and simple interfaces
is treated as a second-class-citizen	parts are treated equally

Table 6.2: Contrasting conformity in software and systems.

In EM and PD modellers and designers share a similar motivation as scientists to seek simple explanations of systems. Although EM and PD are not sciences in the traditional sense they do involve understanding the nature of objects and using this knowledge to simulate and build systems. The interface, representing structure and function, is of little importance in this activity. EM and PD (and natural sciences) are more concerned with the form and context in combination than the largely artificial boundary that separates them [Sim81].

As Brooks points out, the problem of unifying system components is often left to the software developer because the software is typically the last part of the system and is considered more flexible than other more concrete parts. In this way software is treated like a second-class-citizen within the system resulting in software complexity that could perhaps be better accommodated within mechanical and electrical components. This is not so in EM and PD; they are not two-tier development processes. In EM and PD modellers and designers develop descriptions of software and hardware in parallel as they search for the most appropriate, and usually simplest, model and design solution.

### 6.2.3 Changeability

Brooks observes that software is constantly subject to pressures for change (Table 6.3). He concedes that buildings, cars, computers and other systems are also but that these systems are infrequently changed after their initial design; they are superseded by later models that incorporate essential changes in order to meet changes in customer need and technology. Significant changes in the concept of a manufactured automobile are infrequent; changes in the concept of a manufactured computer somewhat less so. In turn, Brooks asserts that changes in both the automobiles and



Software ...	Systems ...
is always under pressure to change	concepts are seldom changed
embodies the system function	function is determined by its structure
change originates externally	change originates from the system context
change is rapid	change is slow

Table 6.3: Contrasting changeability in software and systems.

computers are much less frequent than modifications to installed software.

Brooks points out that an often cited reason for the changeability of software is that the software of a system embodies its function, and that the function is the part of the system that most feels the pressure for change. The function of a system is the result of interaction between components and the nature of the interaction is determined by the details of those components. Thus, small changes to components in a system result in changes to the system function. In EM and PD the focus is on the high-level concept of the system, at least in the early stages, rather than the details of components. So, although the concept of a system changes in EM and PD, it changes slower than representations of the system function or component details.

Brooks identifies two processes that conspire to cause software to be changed:

- as a software product is found to be useful, people discover novel uses for it;
- software is adapted to take advantage of new technology.

Arguably both these phenomena are less to do with the software, or even the computer that executes the software, and more to do with the context of the computer [Sim81]. EM and PD provide a broader conceptual framework than SD allowing modellers and designers to address issues that surround the software, such as usability and technological development.

Brooks concludes that software is embedded in a cultural matrix of applications, users, laws, and machine vehicles, and that their changes inexorably force changes upon the software. With this in mind it would seem that the context of the software and the computer executing the software is the important issue rather than the software itself. EM and PD is suited to understanding software in context



Software ...	Systems ...
is invisible and unvisualizable	are visible and visualizable
is abstract except for representations	are concrete as are its representations
involves reasoning and language	involve common-sense and pictures

Table 6.4: Contrasting invisibility in software and systems.

and using this knowledge in order to develop software.

#### 6.2.4 Invisibility

Brooks argues that software is invisible and unvisualizable and that this is in contrast to systems that are inherently visible and visualizable (Table 6.4). He goes on to say that geometric abstractions used in building systems (as used in EM and PD) are powerful tools: “The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in geometric abstraction” [Bro87].

Brooks continues by saying that the reality of software is not inherently embedded in space making visualization difficult in SD: “[software] has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software we realize that it suggests no particular set of symbols for representation or conventions for organizing symbols” [Bro87]. Languages in EM are developed with a particular mode of observation in mind [ABCY94c]. In this way the definitive languages in EM are not arbitrary but correspond to a way of observing the world. The difference between the languages of EM and the pictorial-language of PD is that definitive languages have an operational interpretation as well as a real-world meaning.

In concluding his section on the invisibility of software Brooks comments on the effect this essential property of software has on mental processes and communication: “In spite of progress in restricting and simplifying the structures of software,

Attack ...	EM ...
uses off-the-shelf software products	reuses existing artefacts
cuts cost of development	speeds modelling by reuse
utilizes spreadsheets and databases	tool is based on spreadsheet principles
combines use and programming	combines use and modelling

Table 6.5: EM themes associated with buy versus build.

they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication between minds” [Bro87].

### 6.3 Attacks on the essential difficulties of software

As well as identifying the four essential properties of software in [Bro87], discussed in the previous section, Brooks recommends four promising ways to attack the essential difficulties of software - buy versus build, requirements refinement and rapid prototyping, incremental development and great designers - to be included in future approaches to SD. This section considers how these attacks relate to the view a programming as configuring systems in EM and PD and the associated views of the computer as artefact and program as systems configuration.

#### 6.3.1 Buy versus build

Brooks draws attention to the important development in the software industry of off-the-shelf software tools, environments and modules: “Every day it is becoming easier, as more and more vendors offer more and better software products for a variety of applications, for software developers to buy existing software instead of developing it themselves” [Bro87] (Table 6.5).

As was identified in previous chapters, and observed during the lift project, EM and PD are based on the free-trade of modelling and design elements that saves on resources and provides the building-blocks for models and designs. This trade operates at two levels:



- conceptual elements used in design and modelling:
- physical elements used to realize designs.

This free-trade in PD has been noticed by others: “The limits of design are culture-bound: all successful designs rest solidly on specific precedents. Because inventors and designers nearly always devise new combinations of familiar elements to accomplish novel results, links to known technology are inevitably present. The inevitability of the old in the new is no check on originality however. The possible combinations of known elements is subject to endless variation” [Fer92].

Brooks identifies the spreadsheet and simple database systems as perhaps the most powerful general off-the-shelf tools. He argues that these powerful tools, so obvious in retrospect and yet so late appearing, lend themselves to myriad uses, some quite unorthodox. The main software tool used for EM so far is the `tkeden` interpreter. The `tkeden` interpreter shares the same general principles as the spreadsheet [Bey97] and gives the modeller access to scripts rather like a database [BCY94]. These principles have proven their importance in a variety of EM projects, including the lift, OXO [BJ94] (Chapter 2), sailboat [NBY94] (Appendix B), railway [ABCY94c], classroom simulation [Dav96] and VCCS [BBY92] projects.

Brooks believes that the increase in off-the-shelf software tools, environments and modules will blur the distinction between programming and use: “the single most powerful strategy for many organizations today is to equip the computer-naïve intellectual workers who are on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs and then to turn them loose” [Bro87]. The EM `tkeden` interpreter and associated definitive languages, such as DoNaLD, ADM, EDEN and SCOUT, aspire to embody the principles underlying such a collection of tools.

### 6.3.2 Requirements refinement and rapid prototyping

Brooks agrees with the widely held belief that the hardest single part of developing software is deciding precisely *what* is wanted: “No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the



Attack ...	EM ...
addresses what is to be built	models what is observed
determines what client wants	involves understanding the subject
involves rapid prototyping	uses visualization and animations
helps conceive the system to be built	parallels natural conceptualization

Table 6.6: EM themes associated with requirements and prototyping.

interfaces to people, to machines, and to other software. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Therefore, the most important function that the software developer performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification. Moreover, the dynamics of a system are hard to imagine. So it is important to plan for extensive interaction between the client and software developer during SD” [Bro87] (Table 6.6).

It was suggested in Chapter 3 that EM might be construed as a process that precedes conventional SD corresponding to requirements engineering, a theme that is taken up later in this chapter. EM is an interactive process during which a representation of the subject is constructed by the modeller. From this process emerges a definitive script that specifies the system structure and function precisely and unambiguously. Such a description is used to simulate the system using the `tkeden` interpreter and forms the basis of further analysis using conventional SD techniques.

Brooks identifies the tools and approaches to rapid prototyping as one of the most important and successful attacks on the essence of software. He defines a system prototype as something that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same hardware speed, size, or cost constraints: “Prototypes typically perform the mainline tasks of the application, but make no attempt to handle the exceptional



Attack ...	EM ...
associates complexity in nature with software	models natural phenomena
advocates top-down approach	begins with high-level concept
results in early working system	maintains up-to-date working model

Table 6.7: EM themes associated with incremental development.

tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability” [Bro87].

The computer model in EM has much in common with the system prototype. It provides some of the behaviour of the subject for the purpose of helping understand the subject. However, the system prototype is typically defined with specific interfaces and functionality in mind whereas in the computer model in EM the interfaces and function are not necessarily preconceived. The modeller is able to step-in as super-agent to resolve problems caused by exceptional tasks or unexpected input and incorporate them into the model on-the-fly.

### 6.3.3 Incremental development - grow don't build software

Brooks gives an account of the history of SD by associating a metaphor for development with each era:

- writing programs;
- building programs (specifications, assembly of components, scaffolding);
- growing programs.

He mentions that the growing metaphor reflects the development of increasingly complex software: “In nature we find constructs whose complexities thrill us with awe. The brain is intricate beyond mapping, powerful beyond imitation, rich in diversity, self-protecting, and self-renewing. The secret is that it is grown, not built” [Bro87] (Table 6.7).

But surely it is better to build rather than grow if the building blocks are available? In EM and PD a model or sketch is built using existing elements if

they are appropriate. This is much less costly in resources than creating elements from scratch. However, when modelling and design elements are not available they have to be synthesized from the basic general concepts of observables, agents and components. Growing a new concept requires ingenuity resulting in something that has an almost mystical quality with the potential for creative discovery, as discussed in previous chapters.

Brooks recommends adoption of the top-down approach to SD: “Mills [Mil71] first proposed that any software should be grown by incremental development. That is, the system should first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then, bit-by-bit, it should be fleshed-out in a step-wise fashion, with the subprograms being fleshed-out in turn. This approach necessitates a top-down approach to design in which each added function and new provision grows out of what is already there” [Bro87].

This top-down approach has its counterpart in EM and PD with the design of a concept followed by the consideration of detail. However, the importance of combining top-down with bottom-up design is identified by Pugh [Pug91, Pug96]. It is no good arriving at a concept that is not cost-effective, or perhaps even impossible, to manufacture. Whilst designing the concept for a system the designer should never lose touch with how the concept is to be realized.

Brooks emphasizes the benefits of having an up-and-running system early on in the SD process by using the top-down approach. In EM the modeller is able to animate high-level concepts of a system. Moreover, the modeller is able to animate low-level elements of a system because of the environment provided by the `tkeden` interpreter. This means that EM can provide the benefits of an up-to-date running program during top-down and bottom-up design.

#### 6.3.4 Great designers

Brooks argues that the central question in how to improve the software art centres on people (Table 6.8). This accords with the importance of the modeller and designer in EM and PD as identified in Chapter 3.

Brooks identifies the importance of methodology but also realizes its limita-



Attack ...	EM ...
centres on people	addresses communication between people
advocates creativity	involves creative exploration
is not method based	has no explicit method
centres on individuals	is essentially a 1-agent activity

Table 6.8: EM themes associated with great designers.

tions (Section 5.4.2): “We can get good designs by following good practices instead of poor ones. Good design practices can be taught. Programmers are among the most intelligent part of the population, so can learn good practice. However, the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a creative process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge” [Bro87].

Brooks points out that “the most exciting breakthroughs have been made by individuals. Although many fine, useful software have been developed by committees and built as part of multipart projects, those software systems that have excited passionate fans are the products of one or a few designing minds, great designers” [Bro87]. There are clearly parallels between individuals designing software and 1-agent modelling in EM.

## 6.4 Software and SD in the future

In a report entitled “Where is Software Headed?” [Lew95] experts in the field of SD from both academia and industry give their predictions for the future of software and SD. This section considers how the views of computer as artefact, program as system configuration and programming as configuring systems in EM and PD relate to their visions of the future of software and SD.

### 6.4.1 Networked computing and concurrency

A general prediction is the continued move towards networked computing: computing with applications, data, and processing power all dispersed across a network. The conceptual framework underlying conventional SD is based on abstract models which make it difficult to describe and reason about such systems. Networks evolved not for any theoretical reason but because use centred around a single processor and data store was found to be impractical. In EM the notion of the computer as a system should make it easier to understand networks and distributed computing.

Simon recognizes the practical and empirical nature of the development of the early timesharing networked systems: “The research that was done to design computer timesharing systems is a good example of the study of computer behaviour as an empirical phenomenon. Only fragments of theory were available to guide the design of a time-sharing system or to predict how a system of a specified design would actually behave in an environment of users who placed their several demands upon it. Most actual designs turned out initially to exhibit serious deficiencies, and most predictions of performance were startlingly inaccurate.

Under these circumstances the main route open to the development and improvement of time-sharing systems was to build them and see how they behaved. And this is what was done. Perhaps theory could have anticipated these experiments and made them unnecessary. In fact they didn’t, and I don’t know anyone intimately acquainted with these exceedingly complex systems who has very specific ideas as to how it might have done so. To understand them, the systems have to be constructed and observed” [Sim81]. Such an approach to development accords with systems development in EM and PD.

Concurrency is closely associated in computing with networked or distributed systems. For a network to work each part must have some independence of operation. So, the prediction of a continued move towards networked computing implies a move towards parallel computing. However, Hill, Larus and Wood [HLW95] point out that the conventional programming model is based on the uniprocessor. They argue for a shared address space model for parallel computation. Central to EM is the notion that variables correspond to actual physical features that all share a



common “address space” within the real-world. Conflicts among changing variables are resolved in EM by understanding the correspondence between variables and observables in the subject. A fuller account of concurrency in EM is given in [BSY88, Nes93, Sla90].

### 6.4.2 Software agents

A central principle of EM is to establish a close correspondence between the computer system and the system perceived or imagined by the software developer. This is achieved by in EM by describing the system in terms of observables and agents. Inevitably, parts of the system acquire names that corresponded to concepts in EM, such as agent and observable. This accords with the predictions of Vetter [Vet95] of the emergence of software agents: distributed computer programs that are capable of carrying out specialized functions on the behalf of humans such as a “knowbot” which intelligently finds information of interest to users over a collection of heterogeneous networked computers. The development of such agents in conventional SD is difficult because the behaviour of the agents depends to a large degree on their environment, which does not generally suit formalization.

### 6.4.3 Object standards and technology

Another general prediction is the continuation of the Object concept into the future of SD. Meyer [Mey95] says of Object technology “it is here to stay”. Consequently there is pressure to make the Object the standard software entity. However, Laplante [Lap95] questions the suitability of the existing Object concept arguing that it is deeply rooted in concepts that evolved in the 1970s with the revolutionary language CLU and in the theories of information hiding attributed to Parnas. This is not to say that the Object is a bad idea, but that it would be worth reviewing the concept in the context of the needs of today. Pree and Pomberger [PP95] argue that establishing standards in Object technology too early could lead to the perpetuation, instead of the solution, of the software crisis. The EM notion of agent offers an alternative to the existing Object concept.

In his book “Object-oriented Software Construction” [Mey88] Meyer presents

a number of standard principles that are traditionally associated with modularity in object-oriented software development:

- the principle of linguistic modular units states that modules must correspond to syntactic units in the language used;
- the principle of few interfaces states that every module should communicate with as few other modules as possible;
- the principle of small interfaces (weak coupling) states that if any two modules communicate at all they should exchange as little information as possible;
- the principle of explicit interfaces states that whenever two modules A and B communicate, this must be obvious from the text of A or B or both;
- the principle of information hiding states that all information about a module should be private to the module unless it is specifically declared public.

In this sense, a module is a more appropriate representation of a software entity than an entity in the real-world. However, texts on object-oriented approaches to SD, including [SM88, SM92] by Shlaer and Mellor, tend to emphasize the direct correspondence between the concept of Object and objects in the real-world. The principles of modularization are seldom observed of entities within the natural world:

- they do not necessarily correspond to a descriptive statement, such as a definition or specification;
- they are not necessarily restricted by the size or number of interfaces they have to other objects (if they can be considered as having interfaces at all);
- they are not necessarily restricted to when they can act;
- they are not necessarily able to make features private.

This suggests that the association between the principles of modularity and real-world objects is inappropriate.

In addition to standards there seems to be a general consensus as to the direction in which Object technology should develop in the future. These include



network stores of Objects to be used in different applications, and specialized Object (horizontal) as well as the traditional application (vertical) development. Pree and Pomberger [PP95] warn that building such higher level standards on the antiquated Object concept will force software developers to produce unnecessarily complicated and unprofessional solutions for problems that could otherwise be solved more efficiently. The EM notion of agent has been adapted by experts in the search for efficient solutions to problems within their own specialist domains, including engineering, SD and education.

Perhaps the main reason why Objects have remained so popular in SD is because they help with the problem of complexity. Laplante says that SD has always been about finding better mechanisms for abstraction to support greater complexity predicting that this trend is set to continue. However, he believes the trend should move away from the use of Objects. Objects achieve their abstraction by generating complex code and relying on complex tools that depend on high-speed modern computers to hide these inefficiencies. Laplante predicts that abstraction will be achieved in the future by tools which harness complex and powerful mental processes to deal with the problem of complexity. Tools based on graphics and real-time interaction instead of formal languages. This principle of providing essentially simple tools and languages that harness the powerful mental processes of the modeller and designer is central to EM and PD.

#### 6.4.4 Product-oriented development

Yet another general prediction is about the move away from the process-oriented approach of conventional SD towards a product-oriented approach in the future. Weide [Wei95] argues that poor design is a major culprit in the software crisis. Many believe it is poor adherence to established engineering processes that is the problem and that this will be improved through proper management. However, Weide makes the point that this assumes that product quality derives largely from process quality. Processes in mature engineering disciplines are of course very important but they came only after successful design had been repeated and observed. EM and PD supports the product-oriented instead of the process-oriented approach by allowing



experts to apply the design knowledge of their own disciplines in developing software.

## 6.5 Requirements engineering in the future

In a report entitled “Requirements engineering: the Emerging Wisdom” [SS96] Siddiqi and Shekaran identify the direction in which requirements engineering is heading. They predict that the next wave of requirements techniques and tools will account for the problem and development context, accommodate incompleteness, and recognize the evolutionary nature of requirements engineering. This section considers how the views of computer as artefact, program as system configuration and programming as configuring systems in EM and PD relate to the future of requirements engineering. Siddiqi is director of the Computing Research Centre and professor of Software Engineering at Sheffield Hallam University. He is also a founding member of the IEEE International Conference on Requirements Engineering and a permanent member of its steering committee. Shekaran has led a variety of research and development efforts in requirements engineering as a manager in Microsoft.

### 6.5.1 Emerging importance of context

The importance of context in requirements engineering is a theme that runs throughout the report by Siddiqi and Shekaran.

Siddiqi and Shekaran point out that increasingly practitioners are realizing the traditional approach to SD analysis, involving the decomposition of the problem into parts and the composition of parts, is not appropriate because the process and parts are situated. They argue that the biggest drawback of this reductionist view of partitioning things into smaller parts is that the context will influence the decomposition.

This limitation of the reductionist view is addressed in the philosophical foundations of EM. Traditional empiricism is essentially reductionist based on the principle that phenomena can be reduced into elements of experience. The philosophical foundations of EM, described in Chapter 2, are embodied in “Radical Em-



piricism.” Radical Empiricism is based on the presumption that the world is a whole, or “conjunction”, with no natural boundaries dividing it into elements: “Conception disintegrates experience utterly” ([Jam96] p.70), “[it] performs on conjunctive relations the usual rationalistic act of substitution - [taking] them not as they are given in their first intention, as parts constitutive of experience’s flow, but only as they appear in retrospect, each fixed as a determinate object of conception, static, therefore, and contained within itself” ([Jam96] p.236). How the world is divided is mostly arbitrary depending on the individual.

Siddiqi and Shekaran introduce the views of Jarke and Pohl [JP94] for whom the juxtaposing of vision and context is at the heart of managing requirements: “[Jarke and Pohl] define requirements engineering as a process of establishing visions in context and proceed to define context in a broader view than is typical for an information-system perspective. Jarke and Pohl partition context into three worlds: subject, usage, and system. The subject represents a part of the outside world in which the system - represented by a structural description - exists to serve some individual or organizational purpose or usage” [SS96].

There are clearly parallels between requirements engineering, in the sense of Jarke and Pohl, EM and PD. At the beginning of this chapter the ideas of form and context were introduced along with the notion of computer as artefact in EM and PD. Jarke and Pohl’s subject, usage and system correspond to context, purpose and form introduced earlier in this chapter with respect to the notion of the computer as artefact. The term subject, as used by Jarke and Pohl, has the same meaning as the term used throughout this thesis and defined in Chapter 3, that is, the object or system being modeled, designed or analyzed.

Siddiqi and Shekaran identify that, whereas in the past most researchers have focused on functional (or behavioural) requirements, the recent trend has been to direct attention to nonfunctional requirements issues. This recent development brings requirements engineering more in line with PD in which the designer has to consider nonfunctional requirements, such as size, weight, ergonomics, documentation and aesthetics, during conceptual design. The similarity between PD and EM identified in previous chapters suggests that EM also deals with nonfunctional



requirements.

For some time now, argue Siddiqi and Shekaran, the SD community has realized the need to broaden its view of requirements to consider the context within which the system will function, with conceptual modelling being the first step: “Borgida, Greenspan, and Mylopoulos’ work [BGM85] on the use of conceptual modelling as a basis for requirements engineering was a major signpost in directing researchers to this perspective” [SS96]. Conceptual modelling and design are central to EM and PD. Both EM and PD involve the process of identifying a high-level concept of the subject and then progressively filling in the detail. This process is sensitive both to the context of the modeller and designer and to the context of the subject.

Siddiqi and Shekaran draw attention to Jackson’s alternative way to look at context [Jac95]: “Jackson faults current SD methods for focusing on the characteristics and structure of the solution rather than the problem. Software, according to Jackson, is the description of some desired machine, and its development involves the construction of that machine. Requirements are about purpose, and the purpose of a machine is found outside the machine itself, in the problem context” [SS96]. Jackson’s views correspond to those in EM and PD: software as a machine description corresponds to the view of a program as a system configuration or representation thereof; development of software as machine construction corresponds to the view of programming as configuring the arrangement of components in a system. The link between purpose and communities within the context has already been mentioned previously in this chapter with respect to the view of the computer, program and programming in EM.

Siddiqi and Shekaran conclude their account of views on requirements engineering with perhaps the most radical of all and yet probably the one that has most in common with EM and PD: “Goguen argues that requirements are information, and all information is *situated* and it is the situations that determine the meaning of requirements. Taking context (or situations) into account means paying attention to both social and technical factors. Focusing on technical factors alone fails to uncover elements like tacit knowledge, which cannot be articulated. Therefore, an



effective strategy for requirements engineering has to attempt to reconcile both the technical, context insensitive, and the social, contextually situated factors.

For Goguen ... requirements emerge from the social interactions between the users and analysts. This goes beyond taking multiple viewpoints and attempting to reconcile them because it does not attempt, *a priori*, to construct some abstract representation of the system. Current methods of eliciting tacit information, such as questionnaires, interviews and focus groups are inadequate, as Goguen points out.

Instead, he advocates “ethnomethodology” [Gog96]. In this approach, the analyst gathers information in naturally occurring situations where the participants are engaged in ordinary, everyday activities. Furthermore, the analyst does not impose so-called “objective” preconceived categories to explain what is occurring. Instead, the analyst uses the categories the participants themselves implicitly use to communicate” [SS96].

There are clearly parallels between Goguen’s view of requirements engineering [Gog94, Gog96, Gog93] and EM and PD: in EM and PD the system begins as a concept within the mind of a single modeller or designer and then is refined into a detailed description of a physical system that can be understood by many. EM is based on the principle of developing languages, such as DoNaLD, SCOUT and ARCA (Chapter 2), that are appropriate for describing particular domains rather than enforcing the use of a single general-purpose language consisting of preconceived concepts. This suggests that an approach to requirements based on the principles of EM and PD would have much in common with the vision of Goguen.

### 6.5.2 End of requirements as contract

Siddiqi and Shekaran argue that the view of the requirement as contract is rapidly becoming outdated: “Most requirements engineering work to date has been by organizations concerned with the procurement of large, one-of-a-kind systems. In this context, requirements engineering is often used as a contractual exercise in which the customer and the software developer organizations work to reach agreement on a precise, unambiguous statement of what the software developer would build.

Trends in the last decade - system downsizing, shorter product cycles, the



increasing emphasis on building reusable components and software architectural families, and the use of off-the-shelf or outsourced software - have significantly reduced the percentage of systems that fit this profile. The requirements-as-contract is irrelevant to most software developers today” [SS96].

The artefacts of EM and PD provide an alternative to the outdated precise unambiguous statement of requirements. In Chapter 3 it was argued that the properties of the statement of requirements - familiarity, unambiguity, explicit meaning, completeness, consistency and convergence - make it ideal for communication and providing a basis for analysis, however, the properties discourage the creativity needed for new systems. The creative properties of the EM and PD artefacts - novelty, ambiguity, implicit meaning, emergence, incongruity and divergence - make them ideal for individuals to model and design the systems of today but make them unsuitable as contracts.

### 6.5.3 Supporting market-driven inventors

Siddiqi and Shekaran identify that the bulk of the software developed today is based on market-driven criteria: “The requirements of market-driven software are typically not elicited from a customer but rather are created by observing problems in specification domains and inventing solutions. Here requirements engineering is often done after a basic solution has been outlined and involves product planning and market analysis. Classical requirements engineering offers very little support for these problems. Only recently have researchers acknowledged their existence” [SS96].

This approach reflects that of EM and PD. The first phase of PD is market analysis during which the product design specification (PDS) is formulated. The PDS acts to constrain the essentially creative phase of conceptual design during which the designer invents a system that satisfies the specification. Similarly, in EM the modeller has an idea of the purpose of a system during modelling. In both EM and PD the model or design of a system is not elicited from somebody else but is instead created by the modeller or designer based on an understanding of the context for the system.



#### 6.5.4 Coping with incompleteness

Siddiqi and Shekaran point out that a complete statements of requirements is a rarely achievable ideal: “One impetus for the switch to the evolutionary development model was the recognition that it was virtually impossible to make all the correct requirements and implementation decisions the first time around. Yet most requirements research agenda continue to emphasize the importance of ensuring completeness in requirements specifications. However, incompleteness in requirements specifications is a simple reality for many practitioners. Goguen echoes this view in his criticism of the prescriptiveness of current methods that insist on complete specifications.” [SS96].

The findings of previous chapters accord with this view of Siddiqi and Shekaran. In Chapter 3 it was shown that the statement of requirement is complete in SD. The requirements are necessarily complete with respect to the models of analysis so that the formal artefacts of SD and the associated methods combine to form a closed system with which the software developer can derive code. However, it is inevitable that the requirements will change during the development of software [SB82] making such a system approach inappropriate. In EM and PD creativity replaces methodology and creative artefacts replace analytical ones. There is no need for the requirement of a system to be complete in EM or PD.

Siddiqi and Shekaran identify the real challenge of coping with incompleteness as how to decide what kinds and levels of incompleteness the software developer can live with: “To this end we need techniques and tools to help determine appropriate stopping conditions in the pursuit of complete requirements specifications - enabling such clarifications to be postponed to a later development stage” [SS96]. In EM and PD the modeller and designer can see the level of detail in an artefact thus allowing them to judge when their representation of a system is complete. Visualization of software by viewing it as a system configuration should allow for similar techniques as in EM and PD with the potential for developing automated tools to help in the task.



### 6.5.5 Integrating design artefacts

Siddiqi and Shekaran point out that software developers need faster ways to conveniently express the problem to be solved and the known constraints on the solution: “Often, getting to [the expression of the problem] fast outweighs the risk of over-constraining the design ... requirements engineering becomes more of a design and integration exercise in this context. We need “wide-spectrum” requirements techniques that can capture and manipulate design-level artefacts, such as off-the-shelf components” [SS96].

This corresponds to the generative phase of EM and PD described in Chapter 5 in which the modeller and designer bring together existing artefacts and synthesize new artefacts. Previous chapters have identified the inherent continuity in EM and PD resulting from the reuse of existing artefacts and parts thereof. Reuse means that generation of artefacts is typically done quickly. Tools such as *tkeden* help in this process by allowing artefacts to be combined and animated without restricting the modeller to preconceived combinations of artefact parts.

Siddiqi and Shekaran identify that there have been very few concrete results to date in providing support for the task of evaluating alternative strategies for satisfying requirements. However, they do note the burgeoning interest and activity in requirements tracing may offer some solutions in the near future. The direct correspondence between subject and representation in EM and PD facilitates tracing of artefact features back to features of the subject. After the generation of a sketch in PD it is evaluated against criteria based on the PDS. Similarly, the artefacts generated in EM are explored with respect to the subject.

### 6.5.6 Making requirements methods and tools more accessible

Siddiqi and Shekaran observe that many practitioners today use general tools like word processors, hypertext links, and spreadsheets for many requirements engineering tasks: “Given the wide variety of contexts in which requirements are determined and systems are built, researchers may be well-advised to focus on specific requirements subproblems and consider building automation support in the form of add-ons to existing general-purpose tools. Less accessible to practitioners are methods that



prescribe a major overhaul of an organization's requirements process and the use of large, monolithic tools" [SS96].

The **tkeden** interpreter in EM embodies the general principles of the spreadsheet. The interpreter improves on the conventional spreadsheet by providing means to define dependencies and the metaphorical representation of variable values in scripts. The modeller is free to extend the basic interpreter by adding more scripts that define underlying algebras for representing the subject within different contexts. The **tkeden** interpreter, and the approach to modelling upon which it is based, has proved accessible to people from various backgrounds. It integrates well into different disciplines, such as engineering and education, and is learned quickly by people in those disciplines. Evidence is in the form of a variety of EM projects in different disciplines, including the lift, OXO [BJ94] (Chapter 2), sailboat [NBY94] (Appendix B), railway [ABCY94c], classroom simulation [Dav96] and VCCS [BBY92] projects.

## 6.6 Conclusion

This chapter has shown that SD can be viewed as systems development. Central to this is the generalization of the concepts of computer, program and programming in SD:

- computer as artefact;
- program as system configuration;
- programming as the process of configuring systems.

An important result of viewing SD as systems development is that EM can be thought of as an approach to developing software. Evidence in support of EM as an approach to developing software is provided in the way of a favourable assessment of how it addresses topical issues in SD and requirements engineering.

There are those who would argue that there is nothing wrong with the conventional view of SD. After all, there are powerful tools and techniques based on the traditional concepts of computer, program and programming in SD:

- computer as an electronic computer;

- program as a sequence of actions stored in a digital computer;
- programming as the process of constructing the sequence of actions in a digital computer.

There is evidence of these techniques and tools being used successfully in industrial software projects [BH95].

The fact remains that the software industry is in crisis despite the use of powerful methods and automated tools in SD. Reports on the software industry, such as those by Gibbs and Jones [Gib94, Jon95], present a bleak picture:

- for every six new large-scale software systems that are put into operation two others are canceled;
- the average SD project overshoots its schedule by half with larger projects doing even worse;
- three quarters of all large systems are termed operating failures which means that either they do not function as intended or are not used at all.

This crisis is not a recent phenomenon. In the autumn of 1968 the NATO Science Committee convened some fifty top academics and industrialists to discuss the growing problem within the software industry. It was decided during this meeting that SD must be turned into an engineering discipline to solve the software crisis. Gibbs observes that, although this realization was made around a quarter of a century ago, software engineering generally remains a term of aspiration.

It might be argued that at least SD has the essential theoretical foundation required for an engineering discipline whereas EM does not. Shaw, cited in [Gib94], argues that engineering disciplines share common stages of evolution. She has observed parallels between software engineering and chemical engineering. Like software developers, chemical engineers try to develop processes to create safe high quality products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modelling, proven design solutions and rigorous quality control methods.



The state of the software industry nevertheless suggests that perhaps the existing theoretically based computer science is not necessarily the right science for industrial SD. Shaw [Gib94] makes the point that, in comparison with established engineering disciplines, software engineering is less mature. She argues that software engineering is more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale projects to the commercial stage she argues that theoretical computer science has yet to build the experimental foundation on which software engineering must rest. EM provides the flexibility for experimentation and the emergence of theories that are appropriate to particular application domains that is arguably lacking in theoretical computer science.

The conceptual framework of computer science has been extended in the past to address the topics outlined previously in this chapter. Indeed, advances over the years based on additions to the traditional concepts of the computer, program and programming have led to breakthroughs in SD [Bro87]:

- high-level languages;
- object-oriented programming;
- artificial intelligence and expert systems;
- program verification;
- “automatic” programming;
- graphical programming;
- environments and tools.

However, Brooks argues that these advances address the *accidental* difficulties of software: “those difficulties that today attend its production but are not inherent” [Bro87]. Promising attacks on the *essential* difficulties of software have more in spirit with EM than SD, as discussed previously in this chapter. Though it is possible that the current paradigm of theoretical computer science could be extended even

further, Kuhn [Kuh70] warns that over-extending a paradigm eventually leads to its collapse and replacement by another more appropriate set of concepts.

It might be argued that the new concepts of computer, program and programming in EM are too radical and signify too great a departure from the traditional paradigm of computer science. Milner identifies the need for a common framework in which to unite many formalisms: “Computer scientists, as all scientists, seek a common framework in which to link and organize many levels of explanation; moreover, this common framework must be semantic, since our explanations are typically in formal language” [Mil93]. Others call for a complete overhaul of the paradigm of computing: “A new paradigm ... must fundamentally change the way we look at problems we have seen in our past. It must give us a new framework for thinking about problems in the future. It changes our priorities and values, changes our ideas about what to pay attention to and what to consider important” [Lie96].

In conclusion, EM can be thought of as an approach to SD so long as the generalized concepts of computer, program and programming are accepted. As the above arguments and counter-arguments indicate, there seems to be no way of predicting whether the new paradigm will be adopted by the SD community. Kuhn argues that changes from an existing paradigm to a rival paradigm depend on the unfathomable social structure of the community and the social processes by which the community is persuaded to adopt the new paradigm [Kuh70]. But whether or not EM is adopted is surely not as important as the need for the SD community to be actively searching for alternative paradigms in case the existing paradigm does not lead to the all-important science that will form the necessary foundation of software engineering. The existing paradigm might evolve into a paradigm that solves the software crisis, but can the industry afford to wait and see?

## 6.7 Limitations of EM for developing software

Having concluded in the previous section that EM can be viewed as an approach to developing software it is important to point out a number of practical limitations of EM in this respect. These limitations are characteristic of EM, and do not necessarily apply to approaches to systems development in general. They are consequences



Unsuccessful	Successful
No historical software measurement data	Accurate software measurement
Failure to use automated estimating tools	Early use of estimating tools
Failure to use automated planning tools	Continuous use of planning tools
Failure to monitor progress in milestones	Formal progress reporting
Failure to use design reviews	Formal design reviews
Failure to use code inspections	Formal code inspections
Generalists used for critical tasks	Specialists used for critical tasks
Failure to use formal configuration control	Automated configuration control
User requirements creep > 35%	User requirements creep < 15%

Table 6.9: Patterns of large software systems: failure and success.

of an approach that emphasizes creativity and generality in systems development.

### 6.7.1 Quality

It is becoming increasingly clear to practitioners that approaches to SD in the future must provide support for quality control [Jon95, Gib94]. It is an empirical fact that testing to find and fix bugs is the most expensive and time-consuming aspect of SD [Jon95, Boe85]. It follows then that the most effective way to reduce the cost and time of software projects is to reduce the number of software defects that reach the test phase of SD. Jones is clear about the importance of quality control: “From a technical point of view, the most common reason for software disasters is poor quality control.” Table 6.9 from [Jon95] shows the direct link between successful software projects and the use of defect prevention planning and pretest defect-removal activities.

EM is limited as an approach to SD because it does not provide support for quality control. EM does not provide techniques for dealing with metrics, using estimating and planning tools, monitoring milestones, formally reviewing and inspecting designs, or controlling configurations. Chapter 2 introduces EM as a means by which the modeller represents their conception of the subject as it evolves [Bey97]. Since the techniques used in quality control assume preconceptions about the subject, embodied in methods and automatic tools, it would be unprincipled to



include such techniques within EM except in the most general form.

Pugh's views on quality control in PD provide a useful insight into the issue of quality control in EM and SD. Pugh points out that quality control in PD is traditionally based on mathematics and detailed knowledge about components. This parallels quality control in SD, indicated in Table 6.9 by the use of formal techniques and the associated low requirements creep (less than 15 percent). Pugh argues that the abstract mathematical models and detailed knowledge about components, required to control quality in PD, does not exist in the case of innovative products. In these cases quality can only be specified in general terms, where imposing quality control can have the undesired effect of producing an unsuccessful conventional design instead of a successful innovative one. This accords with the status of quality and its control within EM.

Pugh clarifies his position by contrasting total design with the Quality Function Deployment (QFD) approach to design that, rather like SD, is based on the customer requirement. The difference, as identified by Pugh, is that total design can be performed without a requirement whereas QFD cannot: "QFD evolution is customer requirement/product driven, while the work described in the design core is driven by more fundamental issues, and can be operated in situations where initially there is no product, and hence no 'voice of the customer' " [Pug91]. Pugh sees QFD as becoming increasingly powerful procedure as the design becomes conceptually static. In the same way, the view of EM as an approach to SD is of a process whereby the customer requirement evolves in parallel with the development of the software so that quality control can play an increasingly significant role as development progresses.

### 6.7.2 Management

It is widely recognized that an improvement in managing software projects has to be made within the software industry. Jones points out that the first six factors in Table 6.9 associated with software disasters are specific failures in the project-management domain, and the next three can be indirectly assigned to poor management practices: "The fact that project-management is the source of so many



problems with software applications means that problems first become visible to customers and upper-management too late for effective damage recovery. Lack of historical measurement of software projects and failure to initially use estimating tools or carefully monitor progress are widespread. This means that projects that get into serious trouble are not identified until very late in development” [Jon95].

EM in this thesis is limited as an approach to SD because it does not provide support for managing the process of development. In other words, EM does not provide support for organizing technical resources and people with the aim of improving the process of development [Pug96]. This thesis has focused on using EM to develop products. By interpreting agents as modellers, designers and software developers EM can be used to model the social and technical context for development. Though this topic is outside the scope of this thesis, concurrent engineering in EM is discussed in these terms in [ABCY94c, ABCY94a, ABCY94b].

Pugh’s views on management in PD provide a useful insight into the issue of management in EM and SD. In Pugh’s model of design it is assumed that the core phases, as described in Chapter 2, are universal, common to all kinds of design and that it is other areas of design activity that give designs their distinctive character [Pug96]. That is to say, different kinds of design may require different kinds of information, techniques and management. Pugh identifies the area of management as of special importance because design activity requires information, resources, and support to be invested in action in the most effective way. This accords with the view of EM being a general approach to systems modelling that is common to many kinds of development including SD.

Pugh’s most recent model, the *business design activity model*, attempts to locate the PD activity firmly within the overall structure of business [Pug96]. The idea is that the design core is constrained not only by the elements of the product design specification - the product design boundary - but also by the elements of the business structure - the business design boundary. If the constraints of the business design boundary are too severe, it will be necessary to take corporate action, restructuring the business to provide designers with more information, more resources, and more support. This notion of management as a context is an appropriate way



to think of the relationship between management and EM.

### 6.7.3 Methodology

The methodical nature of the SD process is paradoxically both its strength and its weakness. This thesis has shown that the conventional methodical approach to developing software discourages creativity. However, when the requirements for a system are stable the methodical approach can be extremely powerful and successful. This accords with the association, shown in Table 6.9, between a low requirements creep (less than 15 percent), the use of automated tools and formal activities. Perhaps the greatest advantage of a methodical approach is that software developers need only be specialists in the SD method and not in particular real-world domains, such as designing lift systems, sailing, playing OXO and constructing jigsaws.

EM is limited in comparison to SD because it is not a method. EM is not a prescribed sequence of actions to be performed by the modeller. Chapter 2 introduces EM as a means by which the modeller represents their conception of the subject as it evolves [Bey97]. Such a process is necessarily iterative in nature and its details are determined by the complex interactions between the modeller and their environment as they learn about the subject (1-agent modelling). Since methods are reconstructions of previous conceptions of subjects, embodied in general techniques and automatic tools, it would be unprincipled to include methods within the general scheme of EM.

The lack of methodology in EM would probably discourage many practitioners from adopting it as an approach to developing software. However, there are those who believe there has been a general over-emphasis on methodology:

- Kaplan warns that, by pressing methodological norms too far, we may inhibit bold and imaginative adventures of ideas [Kap64] (5.4.2);
- Siddiqi and Shekaran predict a shift away from the requirements as the basis for methodical transformation into code towards creating requirements by observing problems in particular domains and inventing solutions [Sid94];
- Milner argues that the general belief that all systems have to be designed



within the rich conceptual frame of an existing methodology is wrong and that new methods can be discovered experimentally through building systems [Mil86].

These views accord with the notion that EM can be performed without a method and that methods emerge through doing EM that are specific to particular domains with their own conceptual frameworks.

There are parallels to be drawn between EM and PD with respect to methodology. Part of the success of the Pugh's model of design is that it provides a guide to design rather than prescribes how design should be done: "I regard the model's structure as being analogous to a child's climbing frame: it provides the framework on which to climb, it imparts confidence and safety, yet it doesn't prescribe or predetermine the methods by which the child gets to the top of the frame or indeed around inside it" (Pugh [Pug91] p.50). This accords with the view of EM as a framework for systems development rather than a prescriptive method.

#### 6.7.4 Scale

Future approaches have to scale up to address the problem of SD in large-scale projects. Jones' findings show that most small software projects are successful, but that risks and hazards of cancellation and major delays rise quite rapidly as the application size increases: "the development of large applications in excess of 5,000 function points [or approximately 500,000 source code statements in a procedural programming language] is one of the most hazardous and risky business undertakings in the modern world" [Gib94].

EM is limited as an approach to SD because it does not scale up to large projects. One reason for this limit to scaleability is technical: visualizations and animations have to be simple given the current computer and `tkeden` interpreter technology. Although alternative technologies are being considered it seems that this limitation will always exist if the desired flexibility of the EM tools is to be kept. Another reason for the limit to scaleability is to do with the principles of EM: the modeller must be able to perceive the correspondence between the artefacts and subject. Since this correspondence is central to EM it would be unprincipled to have

artefacts that were incomprehensible because of their size and complexity.

The relationship between EM and SD is similar to the relationship between conceptual and detail design in PD with respect to scalability. There is clearly a difference between the sketch of a bridge produced during conceptual design and the drawings for the construction of the bridge. The sketch is much simpler and can be easily comprehended by the designer whereas the final drawings are orders of magnitude more complex and typically incomprehensible except by analysis. Moreover, the simple conceptual sketch is essential to the eventual detailed description and construction of the bridge. The process and artefacts of EM can be thought of as the conceptual design and sketch in PD. The method and artefacts of SD can be thought of as the techniques of analysis and the detailed drawing in PD.



## Chapter 7

# Conclusions and further work

This chapter draws conclusions from the discussions and results in Chapters 3,4,5 and 6 and addresses the aims given in Chapter 1 and finishes with suggestions for further research in the area of creative software development.

### 7.1 Conclusions

This thesis has investigated the suitability of EM as a framework for a new approach to SD, that has creative as well as analytical components. This investigation strongly suggests that EM does provide a suitable framework for a new creative approach to SD that combines concepts and principles from EM, PD and software development.

In Chapter 3 the disciplines of EM, PD and SD were compared. It was found that the activity of EM corresponded to conceptual design and that the activity of SD corresponded to conceptually static design in the sense of Pugh [Pug91]. This suggests that SD is associated with the advanced stages of EM when the modeller has committed to a set of artefacts that represent the subject. It was also found that, with respect to subjects, constraints, environments, artefacts, changes and knowledge, there were more similarities between EM and PD than between EM and SD. This suggests that there are some fundamental differences between EM and conventional SD.

Chapter 3 challenges the assumption that the software crisis will be resolved with the emergence of a software engineering discipline that takes an analytical ap-

proach to SD. In the chapter it was suggested that conventional SD is similar to conceptually static design. Pugh [Pug91] states that this kind of design tends to be engineering based suggesting that a software engineering discipline might well emerge from the existing approach to SD. However, Pugh [Pug91] argues that engineering based design can only be commercially successful if the product is conventional, such as in the design of the automobile. Since modern computer systems are typically of an innovative nature this would suggest that more is needed than just a software engineering discipline to solve the software crisis. Pugh [Pug91] argues that engineering has to be seen in a broader creative context if innovative products are to be successfully designed and produced. The primary aim of this thesis is to present EM as a framework for just such a creative context for software engineering.

In Chapter 4 the artefacts of EM, PD and SD were compared to determine similarities and differences between them. This was done by searching for the properties essential for creativity as identified by Finke *et al* [FWS92] in their investigation of creative cognition. It was found that the properties of EM and PD artefacts made them more suited to creative discovery than the artefacts of SD. It was also found that the artefacts of EM and PD had properties similar to the formal models of SD. This suggests that the artefacts of EM can have a mixture of properties to do with creativity and analysis.

In Chapter 5 the actions of EM, PD and SD were compared to determine similarities and differences between them. This was done by reconstructing activities in terms of generative and exploratory actions. It was found that the activities of EM and PD could be reconstructed in terms of actions of both kinds. This suggests that EM and PD are creative activities. Although the activities of SD could also be reconstructed in terms of generative actions they were far more constrained and there were no exploratory actions found in the activities of SD. This suggests that there is a restricted form of creativity when generating artefacts in SD making creative exploration of the resulting artefacts ineffective and unnecessary.

Chapters 3, 4 and 5 investigated how EM relates to an essentially creative discipline such as PD. This was facilitated by using Pugh's notion of total design [Pug91]. The results of the chapters indicate that EM is a suitable framework for



creativity: it was found that EM corresponds to the conceptual design stage of conceptually dynamic design, an essentially creative activity, and that the artefacts and actions of EM share the same creative properties as those of PD. The comparison with PD brought attention to the danger of general concepts and principles of EM acquiring an analytical bias by emphasizing SD only. The analytical and creative significance of the concepts and principles of EM are made clear in Chapters 3, 4 and 5 by relating them to PD and SD.

Chapters 4 and 5 make use of an experimental approach to investigating creativity, in relation to the EM framework, that gives scientific integrity to the findings. Creative cognition [FWS92] characterizes the properties of structures and processes essential to creativity. These properties and processes were investigated by the psychologists Finke, Ward and Smith through experimentation. Chapters 4 and 5 give the results of experiments to find the properties and actions, characterized in creative cognition, within the artefacts and activities of EM, PD and SD.

Chapter 6 presents a paradigm for creative SD and assesses the likelihood of it being adopted by those currently involved in SD. The proposal for the new paradigm uses EM as its framework and has the potential of changing the way SD is thought about and performed. It is centred on the generalized notion of a *computer as a system* that is configured by the software developer. In this new paradigm *programming is configuring the system* and the *program is the system configuration*.

## 7.2 Further work

### 7.2.1 Software engineering

There are many conflicting views about software engineering. There is confusion over whether software engineering even exists as a discipline: it is clear from companies advertising for software engineers and universities teaching courses in software engineering that many in industry and academia believe that software engineering already exists but there are others who argue that an engineering discipline has yet to emerge from SD [Gib94]. There is also a lack of consensus over what is meant by software engineering [Ber92] with research papers giving alternative and sometimes



conflicting definitions of the term.

Arguably the main cause of these conflicting views about software engineering is the problem of understanding engineering in terms of SD. The word engineering has always been associated with engineering in PD and established disciplines such as civil, industrial, electronic and mechanical engineering which are to do with the analysis of objects and systems. This makes it difficult to associate the word engineering with SD which is essentially to do with abstract descriptions of structure and function. This lack of understanding has lead to the term software engineering meaning different things to different people.

The creative approach to SD proposed in this thesis promises to provide a way of understanding SD from an engineering perspective. It achieves this in two ways: first, by relating SD and EM to total design thus providing a framework for understanding conventional engineering disciplines and their role in PD. Second, by broadening the notion of SD to include the development of objects and systems that are the subject of conventional engineering disciplines.

This suggests that software engineering is in the same relation to creative SD as engineering design is to total design in the sense of Pugh [Pug91]:

- software engineering is preceded by an essentially creative activity, such as EM, whereby ideas for objects and systems are generated, represented and evaluated;
- software engineering incorporates an activity whereby representations of ideas for objects and systems are analyzed to turn them into detailed component specifications;
- software engineering is an activity whereby existing objects and systems are analyzed to discover their constituent components and the interfacing between components in order to formulate detailed component specifications;
- software engineering is followed by an activity whereby components are produced and combined into finished products to be sold.

Within the context of SD as systems development component specifications are not restricted to abstract definitions of computer processes and data structures. The



components specified could include integrated circuits, resistors, shafts, bearings, concrete beams and door frames.

Software engineering, within the context of creative software development, would have to deal with the analysis of programs, objects, systems and their representations. Conventional SD already provides a way of analyzing a statement of requirements for a desired system and transforming it into code. Computer science provides powerful techniques and tools for analyzing code and formal representations of structure, behaviour and functionality [OG75, Bar85, Pnu86, Hoo91, AO91, MP92a, Man74, Heh84, San88, Bac87, C90]. Traditional engineering disciplines provide powerful techniques and tools for analyzing objects and systems. But currently there seems to be no established framework for uniting the techniques of SD, computer science and traditional engineering disciplines. Further work could include investigating EM as a possible candidate for such a framework.

A key problem is finding an appropriate framework for analysis within the context of creative SD. Two approaches to analysis emerged during the research for this thesis both based on the notion of observation in EM: “A theory of observation is a plausible basis for a principled method of constructing a program model for a reactive system” [BR92]. These approaches involve

- representing the organization of observations of a phenomena described by a script as a single graph showing observations that cannot happen concurrently, observations that can happen at the same time and contexts for observation (see Example 7.1), and
- representing the sequence of observations of a phenomena described by a script as a single graph showing the order in which observations are be made (see Example 7.2).

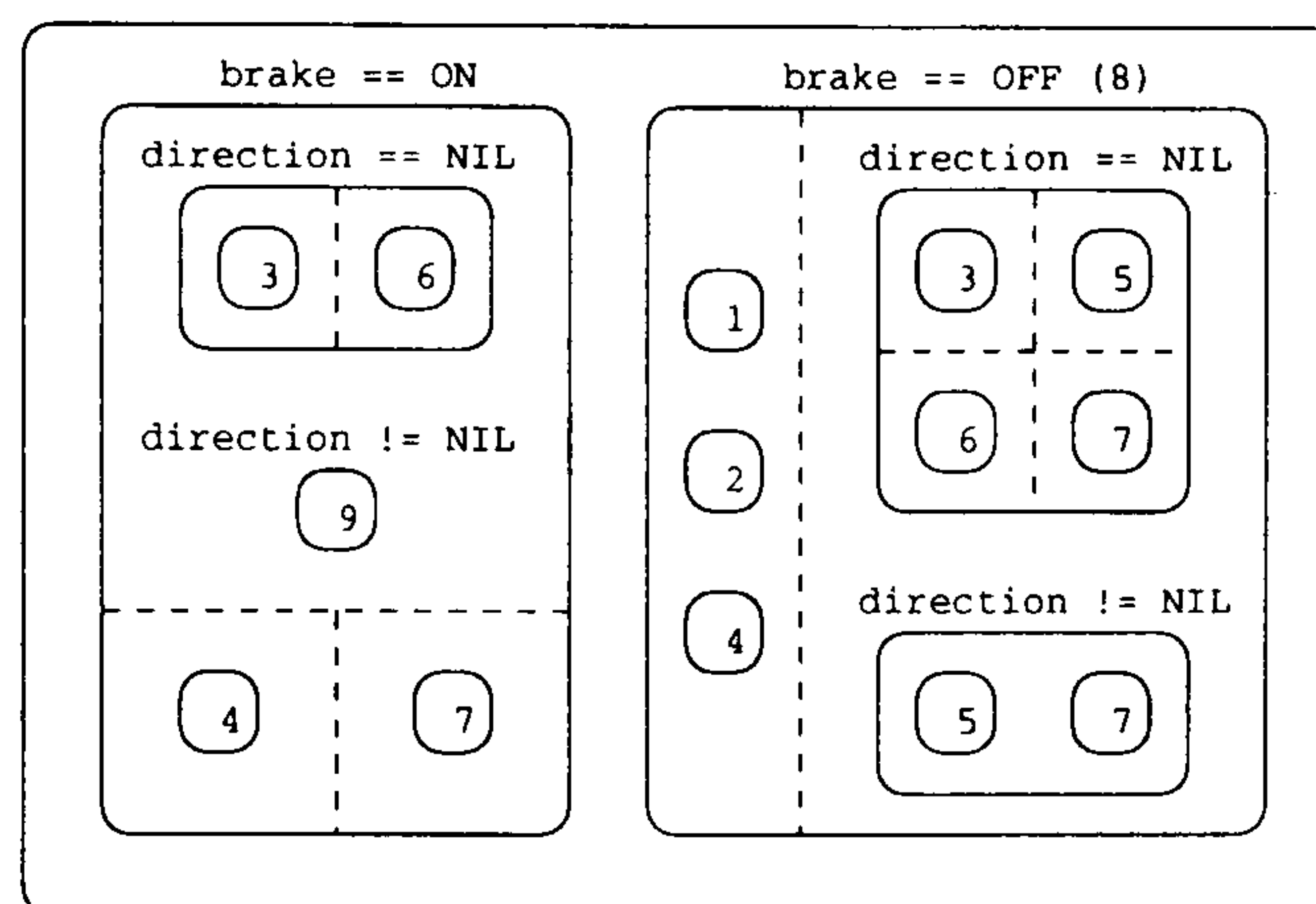
**Example 7.1. Organization of observations in EM.** The guards of the nine MUL ADM actions

```

1 landButton{_F} == UP && _F == floor + 1 && brake == OFF -> brake = ON,
2 landButton{_F} == DOWN && _F == floor - 1 && brake == OFF -> brake = ON,
3 landButton{_F} != OFF && direction == NIL -> destination = _F,
4 floor == _F -> landButton{_F} = OFF
5 carButton{_G} == ON && _G == floor + direction && brake == OFF
                                                                    -> brake = ON,
6 carButton{_G} == ON && direction == NIL -> destination = _G,
7 floor == _G -> carButton{_G} = OFF
8 brake == OFF -> floor = floor + direction,
9 brake == ON && direction != NIL -> brake = OFF

```

corresponding to observations are represented by the graph

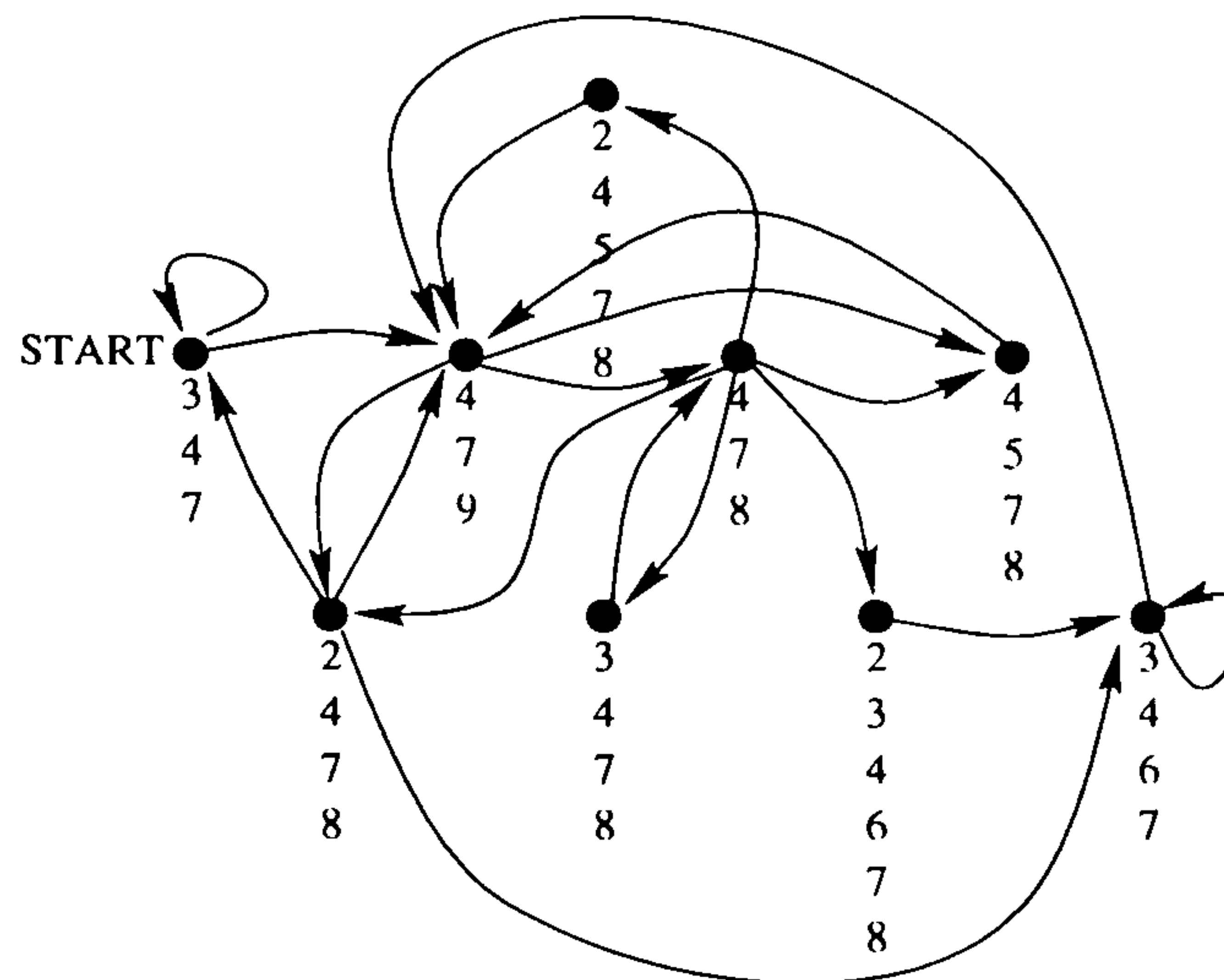


showing observations that cannot happen concurrently in separate boxes, observations that can happen at the same time as boxes separated by a dashed line, and contexts for observation as boxes within boxes.

Other frameworks for analysis in EM are being investigated by Gehring [GYC<sup>+</sup>96]. Further work could include investigating the relation between formal methods and these approaches to analysis. The organization of observations represented in Figure 7.1 is similar to Harel's statecharts [Har88, Har87, HLN<sup>+</sup>88, Har92]. Statecharts and EM artefacts are contrasted by Cartwright and Beynon in [BC95]. Representing behaviour as sequences of states, as in Example 7.2, is standard in computer science especially in the subfield of temporal logic [Har88, Pnu86, MP92a, Hal87].



**Example 7.2. Sequencing of observations in EM.** Some of the sequences of observations described by the the actions of the MUL ADM given above are represented by the graph



showing the order in which observations are made during the interaction by two user entities over a period of ten clock cycles on five different occasions.

### 7.2.2 EM in context

As well as providing the framework for the proposed creative approach to SD EM also has a practical role to play. Within the context of creative SD it would be expected that EM would be used to generate, represent and evaluate alternative ideas for innovative system solutions. However, there is no preconceived strategy given by EM because it is inappropriate to put abstract constraints on situated activities. Further work could involve the investigation of contexts for EM that provide practical support for the generation, representation and evaluation of ideas for system solutions.

Finke *et al* [FWS92] made one of their goals to develop practical techniques for applying the principles of creative cognition in everyday situations. Pugh [Pug91] also makes suggestions for practical techniques to be used in conceptual design:

- brainstorming which is when a group attempts to find a solution to a problem by amassing all the ideas spontaneously contributed by its members;
- exploring new problems and functions suggested by a structure (function-follows-form) instead of the more conventional approach of generating struc-

tures with specific problems or functions in mind (form-follows-function);

- interpreting concepts which are unusual combinations of existing concepts as solutions to problems.

Further work could include finding ways of using such techniques in EM which would use LSD specifications, scripts, visualizations and animations with the aim of helping the modeller.

Pugh [Pug91] talks about creativity in the context of controlled convergence in relation to conceptual design. He argues that creativity should always be carried out in conceptual design within the context of a product design specification (PDS). He suggests an approach based on generation followed by evaluation in which solutions are generated by a designer with the PDS in mind and then these solutions are evaluated by a group using a decision matrix and criteria based on the PDS. Those solutions which pass evaluation continue to be refined and evaluated until the process converges on the best solution. Further work could include investigating a suitable method for controlled convergence for EM perhaps using the PDS and decision matrix as a starting point for research.

### 7.2.3 Distribution of EM tools

The concepts of EM are best communicated by demonstration using the EM tools. This is because their meanings depend on the experiences provided by models generated by EM tools running on computers. So, it is important to have the tools to fully appreciate EM. This presents a dilemma: people need EM tools to fully appreciate the modelling approach and yet they are unlikely to bother acquiring the tools unless they already have an appreciation of EM. The way to break out of this cycle is to make EM tools freely available on the internet and to distribute them to people who might find them useful. This work is important if disciplines that depend on EM, such as creative SD, are to have any hope of adoption in the future. Extensive work in this area has already been done by Yung who most recently developed Tkeden [BSY95]. Currently work in this direction is being done by Cartwright who has ported tools to the PC platform and improved the tools to



make them more appealing and easier to use [BC97]. Most recently a workshop has been organized for teachers with a view to introducing EM into the classroom.

# Bibliography

- [ABCY94a] V. Adzhiev, W.M. Beynon, A. Cartwright, and Y.P. Yung. An agent-oriented framework for concurrent engineering. In *Proc. IEEE Colloquium Issues of Cooperative Working in Concurrent Engineering*, October 1994.
- [ABCY94b] V. Adzhiev, W.M. Beynon, A. Cartwright, and Y.P. Yung. A computational model of multiagent interaction in concurrent engineering. In *Proc. CEEDA94*. Bournemouth University, 1994.
- [ABCY94c] V. Adzhiev, W.M. Beynon, A. Cartwright, and Y.P. Yung. A new computer-based tool for conceptual design. In *Proc. Workshop Computer Tools for Conceptual Design*. University of Lancaster, 1994.
- [ABH86] D. Angier, T. Bissell, and S. Hunt. DoNaLD: a line drawing notation based on definitive principles. Research report 86, Department of Computer Science, University of Warwick, 1986.
- [Ale67] C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1967.
- [And68] B.F. Anderson. *The Psychology Experiment - An Introduction to the Scientific Method*. Brooks-Cole, 1968.
- [AO91] K.R. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.
- [Asp90] W. Aspray. *John von Neumann and the Origins of Modern Computing*. MIT Press, 1990.



- [Bac87] R.J.R. Back. A calculus of refinement for program derivation. Technical report, Abo Akademi, Finland, 1987.
- [Bar85] H. Barringer. A survey of verification techniques for parallel programs. In *LNCS 191*. Springer-Verlag, 1985.
- [BBY92] W.M. Beynon, I. Bridge, and Y.P. Yung. Agent-oriented modelling for a vehicle cruise control system. In *Proc. ESDA92*, 1992.
- [BC95] W.M. Beynon and R. Cartwright. Empirical modelling principles for cognitive artefacts. In *Proc. IEE Colloquium: Design Systems with Users in Mind: The Role of Cognitive Artefacts*, December 1995.
- [BC97] W.M. Beynon and R.I. Cartwright. Empirical modelling principles in application development for the disabled. In *Proc. IEE Colloquium Computers in the Service of Mankind: Helping the Disabled*, March 1997.
- [BCY94] W.M. Beynon, A. Cartwright, and Y.P. Yung. Databases from an agent-oriented perspective. Research report 278, Department of Computer Science, University of Warwick. January 1994.
- [Ber92] D.B. Berry. Academic legitimacy of the software engineering discipline. Research report, Software Engineering Institute, November 1992.
- [Bey86a] W.M. Beynon. ARCA - a notation for displaying and manipulating combinatorial diagrams. Research report 78, Department of Computer Science, University of Warwick, 1986.
- [Bey86b] W.M. Beynon. The LSD notation for communicating systems. Research report 87, Department of Computer Science, University of Warwick, 1986. Presented at 3rd BCTCS, Leicester 1987.
- [Bey89] W.M. Beynon. Definitions as a framework for design. In *Proc. 3rd Eurographics ICAD Workshop*. CWI Amsterdam, 1989.

- [Bey92] W.M. Beynon. Programming principles for the semantics of the semantics of programs. Research report 205, Department of Computer Science, University of Warwick, February 1992.
- [Bey97] W.M. Beynon. Empirical modelling for educational technology. In *Proc. 2nd International Conference on Cognitive Technology. University of Aizu, Japan*, pages 54 – 68. IEEE, August 1997.
- [BFY93] W.M. Beynon, M. Farkas, and Y.P. Yung. Agent-oriented modelling for a billiards simulation. Research report 260, Department of Computer Science, University of Warwick, December 1993.
- [BGM85] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements engineering. *Computer*, pages 82 – 91, April 1985.
- [BH95] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, pages 34 – 41, July 1995.
- [BJ94] W.M. Beynon and M. Joy. Computer programming for noughts and crosses: New frontiers. In *Proc. PPIG94*. Open University, January 1994.
- [BN95] W.M. Beynon and P. Ness. Empiricism in computer-based modelling. In *Proc. 11th BCTCS*, 1995.
- [BNR<sup>+</sup>89] W.M. Beynon, M.T. Norris, S.B. Russ, M.D. Slade, Y.P. Yung, and Y.W. Yung. Software construction using definitions: An illustrative example. Research report 147, Department of Computer Science, University of Warwick, September 1989.
- [BNR95] W.M. Beynon, P. Ness, and S. Russ. Worlds before and beyond words. Research report 331, Department of Computer Science, University of Warwick, 1995. Presented at VF95.
- [Boa84] B. Boar. *Application Prototyping*. Addison-Wesley, 1984.



- [Boe85] B. Boehm. A spiral model of software development and enhancement. In *Proceedings of an International Workshop on Software Process and Software Environments*, August 1985.
- [Boo86] G. Booch. Object-oriented development. *IEEE Software Engineering Transactions*, 12(2), 1986.
- [Boo93] G. Booch. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1993.
- [BR92] W.M. Beynon and S. Russ. The interpretation of states: a new foundation for computation? In *Proc. PPIG92*. University of Loughborough, January 1992.
- [BR94] W.M. Beynon and S. Russ. Empirical modelling for requirements. Research report 277, Department of Computer Science, University of Warwick, September 1994.
- [BRJ98a] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. Estimated publication date Summer, 1998.
- [BRJ98b] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1998. Estimated publication date Spring, 1998.
- [Bro86] H. Brown. *The Wisdom of Science: Its Relevance to Culture and Religion*. Cambridge University Press, 1986.
- [Bro87] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, April 1987.
- [Bro95] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.
- [BRS<sup>+</sup>89] W.M. Beynon, S.B. Russ, M.D. Slade, Y.P. Yung, and Y.W. Yung. Definitive principles and the specification of software. Research re-

port 146, Department of Computer Science, University of Warwick, September 1989.

- [BRY90] W.M. Beynon, S. Russ, and Y.P. Yung. Programming as modelling: New concepts and techniques. In *Proc. ISLIP90*. Computing and Information Science Department, Queen's University, Canada, 1990.
- [BSY88] W.M. Beynon, M.D. Slade, and Y.W. Yung. Parallel computation in definitive models. In *Proc. CONPAR88*. Department of Computer Science, University of Warwick, June 1988.
- [BSY95] W.M. Beynon, C.J. Sidebotham, and Y.P. Yung. Computer-assisted jigsaw construction: a case-study in empirical modelling. In *Proc. 5th Eurographics Workshop on Programming Paradigms for Graphics*, September 1995.
- [BYCH92] W.M. Beynon, Y.P. Yung, A.J. Cartwright, and P.J. Horgan. Scientific visualization: Experiments and observations. In *Proc. Eurographics Workshop Visualization in Scientific Computing*, 1992.
- [C90] Morgan C. *Programming from Specifications*. Prentice-Hall, 1990.
- [Car94] A. Cartwright. *Application of Definitive Scripts to Computer-Aided Conceptual Design*. PhD thesis, Department of Engineering, University of Warwick, July 1994.
- [CM81] E. Conrad and T. Maul. *Introduction to Experimental Psychology*. John Wiley and Sons, 1981.
- [CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1990.
- [Dav93] A.M. Davis. *Software Requirements - Objects, Functions and States*. Prentice-Hall, 1993.
- [Dav96] E.L. Davis. Modelling human interaction, 1996. Bachelor's dissertation.



- [DeM78] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.
- [Dep92] Department of Computer Science, University of Warwick. *Technical Document for the Scout System*, October 1992.
- [Deu88] M.S. Deutsch. Focusing real-time systems analysis on user operations. *IEEE Software*, September 1988.
- [Deu89] M.S. Deutsch. Enhancing testability with scenario oriented engineering. In *6th Int. Conf. on Testing Computer Software*, 1989.
- [Dro89] R.G. Dromey. *Program Derivation, the Development of Programs from Specifications*. Addison-Wesley, 1989.
- [DSWW93] D. Dickson, E. Shimmin, P. Wilson, and C. Wood. Experiences with applying the Shlaer-Mellor methodologies. Technical report, IBM WSDL, 1993.
- [DT90] M. Dorfman and R.H. Thayer. *Standards, Guidelines, and Examples on System and Software Requirements Engineering*. IEEE Computer Society Press, 1990.
- [EE90] C. Eames and R. Eames. *A Computer Perspective: Background to the Computer Age*. Harvard University Press, 1990.
- [Fer77] E.S. Ferguson. The mind's eye: Nonverbal thought in technology. *SCIENCE*, pages 827 – 197, August 1977.
- [Fer92] E.S. Ferguson. *Engineering and the Mind's Eye*. The MIT Press, 1992.
- [FHRK93] M.E. Fayad, L.J. Hawn, M.A. Roberts, and J.R. Klatt. Using the shlaer-mellor object-oriented analysis method. *IEEE Software*, March 1993.
- [FK92] R.G. Fichman and C.F. Kemerer. Object-oriented conventional analysis and design methodologies. *Computer*, pages 22 – 39, October 1992.

- [FS97] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, 1997.
- [FWS92] R.A. Finke, T.B. Ward, and S.M. Smith. *Creative Cognition: Theory, Research and Applications*. The MIT Press, 1992.
- [Gib94] W.W. Gibbs. Software's chronic crisis. *Scientific American*, pages 73 – 81, September 1994.
- [Gog93] J. Goguen. Social issues in requirements engineering. In *Proc. Intl. Conference Requirements Engineering*, 1993.
- [Gog94] J. Goguen. Requirements engineering as the reconciliation of technical and social issues. In *Requirements Engineering: Social and Technical Issues*. Academic, 1994.
- [Gog96] J. Goguen. Formality and informality in requirements engineering. In *Proc. Intl. Conference Requirements Engineering*, 1996.
- [Goo90] D. Gooding. *Experiment and the Making of Meaning*. Kluwer, 1990.
- [Gre70] R.L. Gregory. *The Intelligent Eye*. Weidenfeld and Nicolson, 1970.
- [Gre94] R.L. Gregory. *Eye and Brain, the Psychology of Seeing*. Oxford University Press, 1994.
- [GS83] D. Gentner and A.L. Stevens. *Mental models*. Hillsdale, 1983.
- [GYC<sup>+</sup>96] D.K. Gehring, S. Yung, R.I. Cartwright, W.M. Beynon, and A.J. Cartwright. Higher-order constructs for interactive graphics. In *The proceedings of the Eurographics UK Chapter, 14th Annual Conference, 1996*, 1996.
- [Hal87] R. Hale. Using temporal logic for prototyping: The design of a lift controller. In *LNCS 398 Temporal Logic in Specification*. Springer-Verlag, 1987.
- [Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11 – 19, September 1990.



- [Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231 – 274, July 1987.
- [Har88] D. Harel. On visual formalisms. *ACM Comms.*, pages 514 – 530, May 1988.
- [Har92] D. Harel. Biting the silver bullet: Towards a brighter future for software development. *IEEE Computer*, January 1992.
- [Heh84] E.C.R Hehner. *The Logic of Programming*. Prentice-Hall, 1984.
- [HF75] R.N. Haber and A.H. Fried. *An Introduction to Psychology*. Holt, Rinehart and Winston, Inc., 1975.
- [HLN<sup>+</sup>88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: A working environment for the development of complex reactive systems. In *Proc. 10th Intl. Conference Software Engineering*, April 1988.
- [HLW95] M.D. Hill, J.R. Larus, and D.A. Wood. Portably supporting parallel programming languages. In *Where Is Software Headed?* IEEE Computer, August 1995.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [Hod77] W. Hodges. *Logic: an Introduction to Elementary Logic*. Penguin Books, 1977.
- [Hof93] H.F. Hofman. Requirements engineering - a survey of methods and tools. Research report, Institute for Informatics, University of Zurich, March 1993.
- [Hoo91] J. Hooman. Specification and compositional verification of real-time systems. In *LNCS 558*. Springer-Verlag, 1991.
- [HT95] K.J. Holyoak and P. Thagard. *Mental Leaps: Analogy in Creative Thought*. The MIT Press, 1995.

- [Jac83] M. Jackson. *System Development*. Prentice-Hall, 1983.
- [Jac92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [Jac95] M. Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.
- [Jam96] W. James. *Essays in Radical Empiricism*. Bison Books, 1996.
- [JL83] P.N. Johnson-Laird. *Mental models, towards a cognitive science of language, inference, and consciousness*. Cambridge University Press, 1983.
- [JL88] P.N. Johnson-Laird. *The Computer and the Mind: An Introduction to Cognitive Science*. Harvard University Press, 1988.
- [Jon95] C. Jones. Patterns of large software systems: Failure and success. *IEEE Computer*, pages 86 – 87, March 1995.
- [JP94] M. Jarke and K. Pohl. Requirements engineering in 2001: (virtually) managing a changing reality. *Software Engineering*, pages 257 – 266, November 1994.
- [Kap64] A. Kaplan. *The Conduct of Inquiry*. Chandler, 1964.
- [Kuh70] T.S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1970.
- [Lam88] D.A. Lamb. *Software Engineering*. Prentice-Hall, 1988.
- [Lan93] N. Lang. Shlaer-Mellor object-oriented analysis rules. *Software Engineering Notes*, January 1993.
- [Lap95] P. Laplante. A retrospective look forward. In *Where Is Software Headed?* IEEE Computer, August 1995.
- [Lew95] T. Lewis. Where is software headed? *IEEE Computer*, pages 20 – 32, August 1995.



- [Lie96] H. Lieberman. Intelligent graphics. In *New Paradigms for Computing*. Communications of the ACM, August 1996.
- [LK95] P. Loucopoulos and V. Karakostas. *System Requirements Engineering*. McGraw-Hill, 1995.
- [M<sup>+</sup>88] C. Morgan et al. On the refinement calculus. Research report PRG-70, Oxford University Computing Laboratory, October 1988.
- [Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
- [Mar90] J. Martin. *Information Engineering*. Prentice-Hall, 1990.
- [MEGT96] D. Morris, G. Evans, P. Green, and C. Theaker. *Object-Oriented Computer Systems Engineering*. Springer-Verlag, 1996.
- [Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.
- [Mey95] B. Meyer. From process to product: Where is software headed? In *Where Is Software Headed?* IEEE Computer, August 1995.
- [Mil56] G.A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review* 63, March 1956.
- [Mil71] H.D. Mills. Top-down programming in large systems. *Debugging Techniques in Large Systems*, 1971.
- [Mil86] R. Milner. Is computing an experimental science?, 1986. Inaugral lecture of the Laboratory for Foundations of Computer Science, University of Edinburgh.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [Mil93] R. Milner. Elements of interaction. *Comm. ACM*, 36(1):79 – 97, 1993. Turing Award.
- [MK97] P. Moin and J. Kim. Tackling turbulence with supercomputers. *Scientific American*, January 1997.

- [MP92a] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
- [MP92b] D.E. Monarchi and G.I. Puhr. A research typology for object-oriented analysis and design. *ACM Comms.*, pages 35 – 47, September 1992.
- [Nar93a] R. Narasimhan. *Software Industry: a Developing Country Perspective*, pages 18 – 26. United Nations Industrial Development Organization, Vienna, 1993.
- [Nar93b] B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.
- [NBY94] P.E. Ness, W.M. Beynon, and Y.P. Yung. Applying agent-oriented design to a sail boat simulation. In *Proc. ESDA94*, 1994.
- [Nes93] P.E. Ness. The parallel execution of definitive programs. Master's thesis, Department of Computer Science, University of Warwick, 1993.
- [Nie89] O. Nierstrasz. A survey of object-oriented concepts. In *Object-Oriented Concepts, Databases and Applications*. ACM, 1989.
- [Nor91] D.A. Norman. *Cognitive Artifacts*. Cambridge University Press, 1991.
- [NS76] A. Newell and H.A. Simon. Computer science as empirical inquiry: Symbols and search. *Comm. ACM*, 19(3):113 – 126, 1976. Turing Award.
- [OG75] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, pages 319 – 340, November 1975.
- [Per95] M. Perry. Cognitive artefacts and collaborative design. In *Proc. IEE Colloquium: Design Systems with Users in Mind: The Role of Cognitive Artefacts*, December 1995.
- [Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *LNCS 224 Current Trends in Concurrency*. Springer-Verlag, 1986.



- [Poh96] K. Pohl. Requirements engineering: An overview, 1996. In *Encyclopedia of Computer Science and Technology*, Volume 36. Marcel Dekker Inc., New York.
- [PP95] W. Pree and G. Pomberger. The past as prologue. In *Where Is Software Headed?* IEEE Computer, August 1995.
- [Pug91] S. Pugh. *Total Design: Integrated Methods for Successful Product Engineering*. Addison-Wesley, 1991.
- [Pug96] S. Pugh. *Creating Innovative Products Using Total Design: the Living Legacy of Stuart Pugh*. Addison-Wesley, 1996.
- [R<sup>+</sup>91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [RGE<sup>+</sup>94] T.E. Rothenfluh, J.H. Gennari, H. Eriksson, et al. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTEGE-II solutions to Sisyphus-2. In *Proc. Knowledge Acquisition Workshop*, 1994.
- [Roy70] W.W. Royce. Managing the development of large systems: Concepts and techniques. In *Proceedings of WesCon*, August 1970.
- [Rum93] J. Rumbaugh. Objects in the twilight zone. *JOOP*, pages 18 – 23, 1993.
- [Rus97] S.B. Russ. Empirical modelling: The computer as a modelling medium. In *BCS Computer Bulletin*, April 1997.
- [sai63] The glenans sailing manual, 1963.
- [Sai96] H. Saiedian. An invitation to formal methods. *IEEE Computer*, pages 17 – 30, April 1996.
- [San88] D. Sannella. A survey of formal software development methods. Research report, University of Edinburgh, July 1988.

- [SB82] W. Swartout and R. Balzer. The inevitable intertwining of specification and implementation. *Comms. ACM*, July 1982.
- [She96] W. Sheeran. Working knowledge: Vertical safety. *Scientific American*, page 96, May 1996.
- [Sid94] J. Siddiqi. Challenging universal truths in requirements engineering. *IEEE Software*, pages 18 – 19, March 1994.
- [Sim81] H.A. Simon. *The Sciences of the Artificial*. The MIT Press, 1981.
- [Sla90] M. Slade. Definitive parallel programming. Master’s thesis, Department of Computer Science, University of Warwick, April 1990.
- [SM88] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis - Modeling the World in Data*. Yourdon Press, 1988.
- [SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles - Modeling the World in States*. Yourdon Press, 1992.
- [Smi87] B.C. Smith. Two lessons in logic. *Computer. Intell. vol. 3*, pages 214 – 218, 1987.
- [Smi95] B.C. Smith. The foundations of computation. In *Proc. AISB95 Workshop on the Foundations of Cognitive Science*, April 1995.
- [SS94] C. Sidebotham and L. Suker. An up-lifting journey into definitive programs. Unpublished, September 1994.
- [SS96] J. Siddiqi and M.C. Shekaran. Requirements engineering: The emerging wisdom. *IEEE Software*, pages 15 – 19, March 1996.
- [Sut93] S. Sutherland. Book reviews: More models. *Nature*, May 1993.
- [SW88] R. Sommerhalder and S.C. Van Westerhennen. *The Theory of Computability - Programs, Machines, Effectiveness and Feasability*. Prentice-Hall, 1988.



- [Tur36] A.M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 1936.
- [Vet95] R. Vetter. Softbots, knowbots, and whatnots. In *Where Is Software Headed?* IEEE Computer, August 1995.
- [WBJ90] R. Wirfs-Brock and R.E. Johnson. Surveying current research in object-oriented design. *ACM Comms.*, pages 104 – 124, September 1990.
- [WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.
- [web13] Webster’s revised unabridged dictionary, 1913.
- [Wei95] B.W. Weide. Challenges of software design and the undergraduate computing curriculum. In *Where Is Software Headed?* IEEE Computer, August 1995.
- [Who78] B.L. Whorf. *Language, Thought and Reality*. MIT Press, 1978.
- [Win90] J.M. Wing. A specifier’s introduction to formal methods. *IEEE Computer*, pages 8 – 24, 1990.
- [Wol92] L. Wolpert. *The Unnatural Nature of Science*. Faber and Faber, 1992.
- [YC75] E. Yourdon and L.L. Constantine. *Structured Design*. Yourdon Press, 1975.
- [Yos92] G.R. Yost. Configuring elevator systems. Research report, Digital Equipment Corporation, 1992.
- [Yos94] G.R. Yost. Sisyphus-2: Configuring elevator systems, 1994. <http://www-smi.stanford.edu/projects/protege/sisyphus-2/>.
- [You92] E. Yourdon. *Decline and Fall of the American Programmer*. Yourdon Press, 1992.

- [Yun90] Y.W. Yung. EDEN: An engine for definitive notations. Master's thesis. Department of Computer Science, University of Warwick. September 1990.
- [YY88] Y.P. Yung and Y.W. Yung. *The EDEN Handbook*. Department of Computer Science, University of Warwick, 1988. Updated in 1996.
- [Zie94] R.M. Zielinski. *OOThe Object-Oriented Documentation Tool Manual*. Metod System Utveckling, 1994.



## Appendix A

# Empirical Modelling of a Sailboat

This chapter describes my experiences of constructing a sailboat simulator (SBS). The physical properties of the sail, rig and hull are described within the model. Following the principles introduced in Chapter 2, the state of the SBS is described using a definitive script, and an agent-oriented design method is used to determine and construct each of the sailboat components: sail, rig, hull and sailor. The resulting simulation combines a model of the sailboat dynamics, a simple graphical animation and an interface through which the user can play the role of the sailor.

### A.1 Common-sense knowledge

Sailing and sailboats were already familiar subjects to me before I began constructing the SBS. This knowledge was based on my experience of sailing various craft, including dinghies, sailboards and a yacht. I knew

- how to react to situations as they arose in the boat,
- how to predict situations arising in and around the boat,
- how to devise courses of action to deal with predicted situations, and
- the meaning of sailing terms for giving and understanding instructions.

In other words, I *knew* how to sail. I shared this common-sense [Wol92] sailing knowledge with everybody else who was able to sail. My task was to represent this essentially subjective, practical, vague, inconsistent, situated, analogical [HT95] and phenomenological [GS83] knowledge of how a sailboat behaves in the form of a simulation.

## A.2 Agent-oriented modelling

I found that the concepts of LSD corresponded to my common-sense notions of sailing. I was able to identify four agents:

- the sailor who steers the sailboat and adjusts the position of the sail;
- the sail that is blown by the wind;
- the rig that holds the sail between a boom and mast;
- the hull that holds the rig and is stabilized by a keel.

The sailor agent was the easiest to identify because it was simply me. The other three agents I identified by considering causality between agents. I began with the sail agent and considered what stops the sail from blowing away in the wind. This thought process resulted in the identification of the rig agent. Similarly, by thinking what stops the rig from blowing over I identified the hull agent. This was essentially common-sense thinking involving personification and the notions of causality [Wol92, HT95].

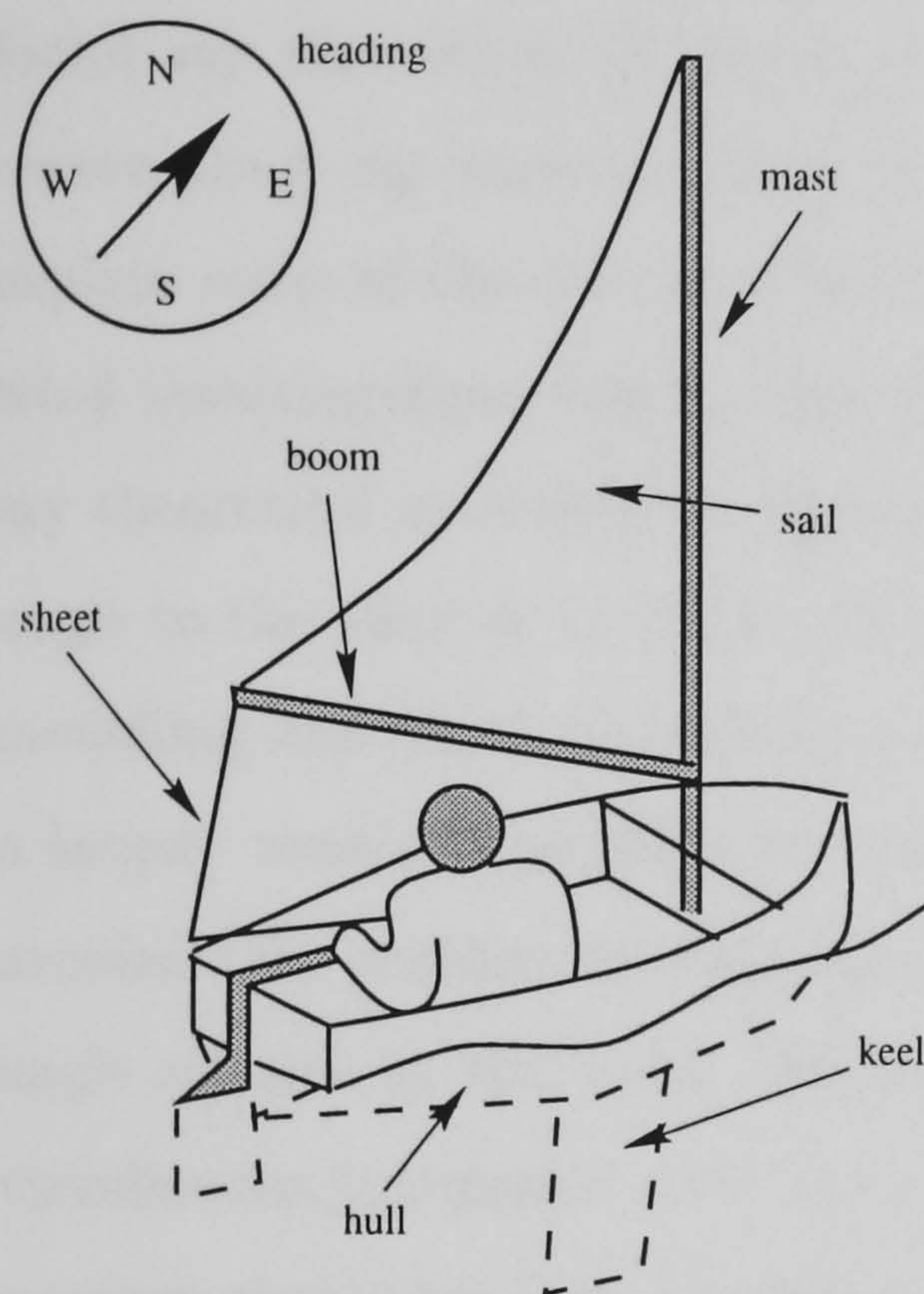
## A.3 Observation-oriented modelling

I continued my LSD specification of the SBS by identifying the observables associated with each agent. The oracles and handles of the sailor were the easiest to identify because I had only to remember what I observed during sailing. Having identified the oracles and handles of the sailor agent, as shown in Example A.1, I then went about forming oracle-handle pairs by attributing oracles and handles to



the sail, rig and hull agents. Some oracles and handles did not make a pair, such as the oracle for wind direction, suggesting an openness about the model.

**Example A.1. Representing the modeller in the SBS.** By defining the sailor agent in LSD I was effectively modelling myself. The diagram shows what I was aware of while sailing.



```
agent sailor {
  oracle
    list
    driving_force
    hull_speed
    wind_dir
    heading
    sheetlenmin
    sheetlenmax
    sheet_len
  handle
    heading
    sheet_len
  derivate
    turn = user_input(turn_type)
    sheetdir = user_input(sheet_type)
  protocol
    turn == starboard -> inc(heading)
    turn == port -> dec(heading)
    (sheetdir == out) &&
    (sheet_len < sheetlenmax) -> inc(sheet_len)
    (sheetdir == in) &&
    (sheet_len > sheetlenmin) -> dec(sheet_len)
}
```

Having defined the oracles and handles for the sailor agent I formed oracle-handle pairs by attributing oracles and handles to the sail, rig and hull agents.

When I came to define the derivates and protocols for the sail, rig and hull agents I had a decision to make. I could either

- represent my common-sense knowledge of causality and agency in sailing as mainly protocol definitions, or
- apply my school-book knowledge of Newtonian mechanics to explain the relations between observables, represent this relation as derivates and test the resulting definition against my common-sense knowledge.

I had the choice between two approaches because of my combined sailing and scientific background. However, if I was a sailor who had not learned about physics



then I would have produced a model based on my common-sense knowledge alone (cf. naive physical models of children [GS83]) suggesting that theory is not necessary for EM. But, the fact remained, that I did know about physics and decided that a more generalized and complete model would result from taking the second choice.

During the definition of the derivatives for the first agent, the sail agent, I found my theoretical knowledge to be insufficient and had to resort to directly representing my common-sense knowledge. Using the idea of vectors helped to explain some of the observations of the sail, such as the angle between the sail and wind resulting from subtracting one from the other. However, I could not apply my theoretical knowledge to explain the driving force of the sail with respect to its angle to the wind or to derive the values of constants. I have since discovered that modelling and simulating aerodynamic effects from first principles has always been a largely unsolved problem in mathematics and physics [Asp90, MK97, sai63]. I avoided this problem by representing my knowledge as a function in terms of sail angle relative to the wind. Example A.2 shows the sail agent definition and the visualization/animation (the visualization and animation are not distinguished by a screen-shot) that I used to test my theory for the sail forces.

By simplifying my model of the dynamics of the hull in the water I was able to define the rig and hull agents almost entirely in terms of physical theories of motion, the only exceptions being the values of constants. The rig and hull agent definitions use integrals to represent the invariant relation between observables over time. For example, the speed of the sailboat at any time is an integral of the acceleration of the boat. This had the effect of introducing time into the model as an observable. Example A.3 shows the rig and hull agent definitions and the visualization/animation that I used to test my theory of sailboat forces.

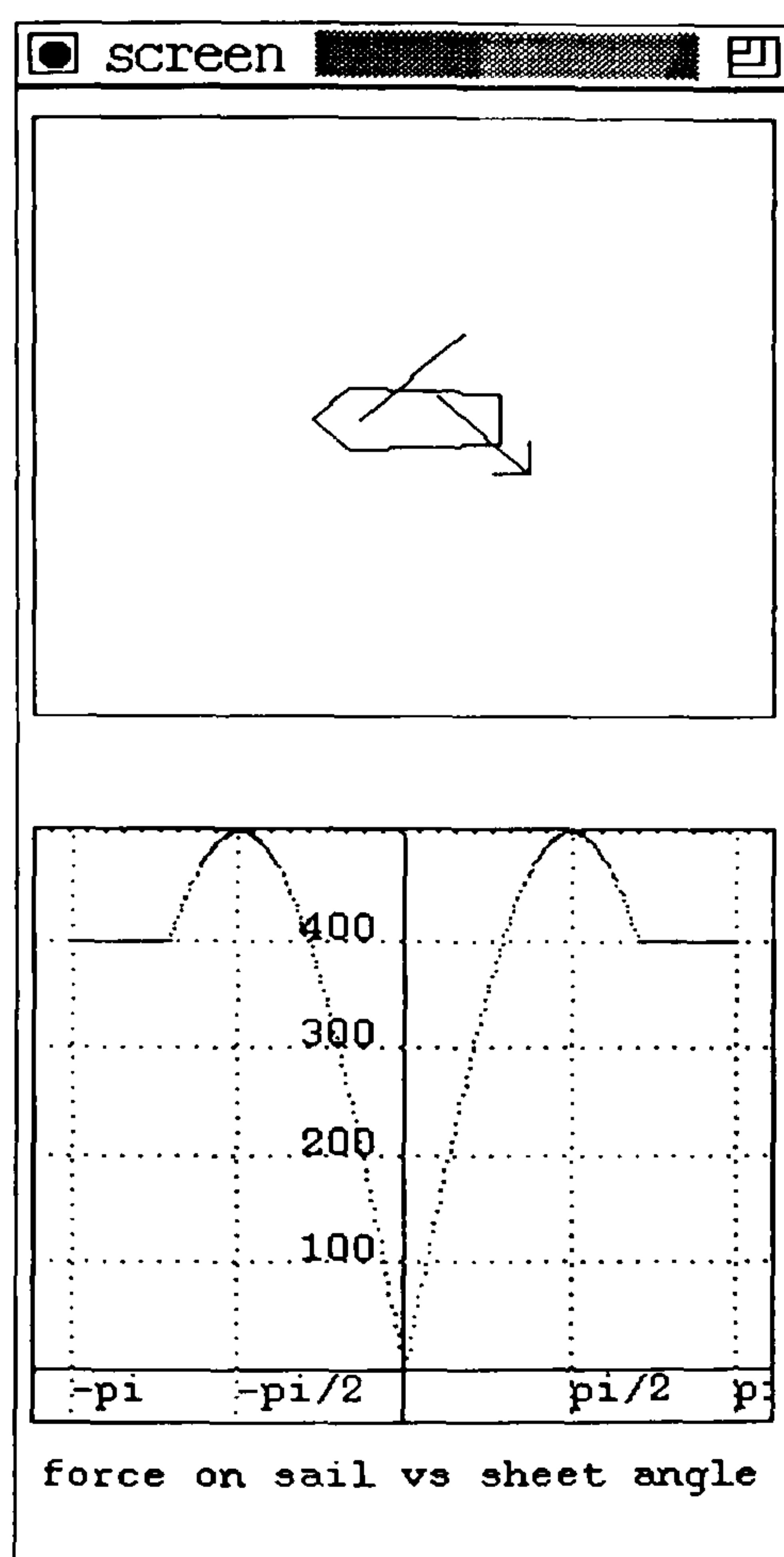
## A.4 Definitive representation of the sailboat

While defining the sail agent I would constantly refer to and modify the visualization/animation, shown in Example A.2, in order to test the emerging theory of the sail forces and discover appropriate values for constants. DoNaLD and SCOUT scripts, defining the image of the sailboat and forces, were written at the start



of modelling and remained essentially the same throughout. I would repeatedly convert the derivatives into definitions in EDEN and experiment with the resulting visualization by changing the sheet variable with the graph, shown in Example A.2, emerging over time. In order to save time I defined an EDEN action that changed the sheet observable automatically thus animating the visualization.

**Example A.2. Representing the sail in the SBS.** I defined the sail as an LSD agent and then tested the specification using the combined visualization and animation shown in the diagram.



```
agent sail {
  const
    drivingFmin = 400.0 // minimum driving force [N]
    drivingFmax = 500.0 // maximum driving force [N]
    k = asin(drivingFmin/drivingFmax)
  state
    sail_dir      // sail direction as bearing [rad]
    sail_wind     // angle between sail and wind [rad]
    driving_dir   // sail driven anti/clockwise [1/-1]
    driving_force // driving force of sail [N]
  oracle
    heading
    wind_dir
    sheet
  derivate
    sail_dir = heading + sheet
    sail_wind = wind_dir - sail_dir
    driving_dir = (sin(sail_wind) < 0.0) ? 1 : -1
    driving_force is
      (cos(sail_wind) < -cos(k)) ?
        drivingFmin : abs(sin(sail_wind))*drivingFmax;
}
```

I had to experiment with the simulation in order to find the appropriate representation for the driving force and values for constants.

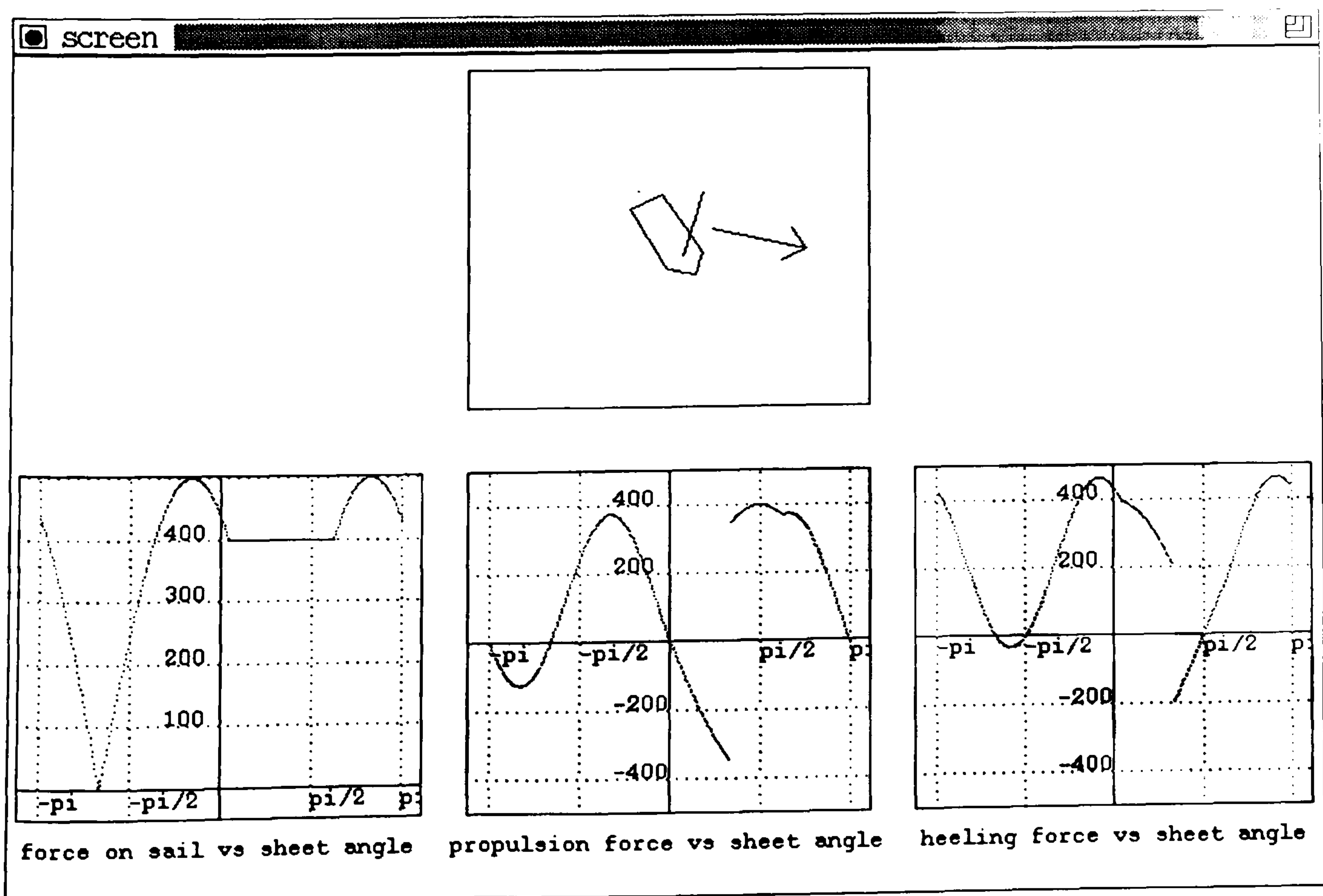
The visualization/animation of the sail was extended to include the graphs of propulsion and heeling force experienced by the sailboat, as shown in Example A.3. The new screen image was simply created by appending a script defining the new graphs to the existing DoNaLD and SCOUT scripts. The derivatives were converted into definitions except for the integral derivatives that were implemented using EDEN actions, as shown in Example A.3. The new visualization/animation was used in the same way as the original to test the emerging theory of sailboat forces.

**Example A.3. Representing the rig and hull in the SBS.** I defined the rig and hull agents in LSD and tested the specifications using the visualization/animation shown in the diagram.

```

agent rig {
const
    boom_len = 2.5 // length of boom [m]
    mast_len = 4.0 // height of mast [m]
    rig_moi = 200.0 // moi of rig [kg m^2]
    resistK = 50.0 // friction constant
state
    sheet          // angle between keel and sail [rad]
    sheet_set      // setting of sheet [rad]
    sheet_len      // length of sheet [m]
    sailT          // torque of sail about mast [Nm]
    resistT        // dampening torque [Nm]
    sailAacc       // angular acc of sail [rad/s^2]
    sailAvel       // angular vel of sail [rad/s]
oracle
    driving_force
    driving_dir
    t              // time [s]
handle
    sailAvel
derivate
    sailT = driving_force * -driving_dir * boom_len / 2.0
    resistT = resistK * -sailAvel
    sailAacc = (sailT + resistT) / rig_moi
    sailAvel = integ_wrt(sailAacc,t)
    sheet = integ_wrt(sailAvel, t)
    sheet_set = 2 * acos(sqrt(1 - (sheet_len*sheet_len)
                                / (4*boom_len*boom_len)))
}

```





```

agent hull {
const
    dragK = 100.0      // drag coefficient of water
    boat_mass = 400.0  // mass of boat [kg]
    ballast_mass = 100.0 // mass of ballast [kg]
    ballast_weight = ballast_mass * g // weight of ballast [N]
    keel_depth = 1.5    // depth of keel holding ballast [m]
    hull_moi = 600.0     // moment of inertia of hull [kg m^2]
    dampK = 500.0       // resistance coefficient to hull listing
state
    hull_speed          // boat speed in water [m/s]
    list                // angle of boat from vertical [rad]
    drag                // drag of boat in water
    forward_force       // forward force of boat [N]
    acceleration        // acceleration of boat [m/s^2]
    hull_speed          // speed of boat [m/s]
    side_force          // sideways force of boat [N]
    sailTq              // torque of sail about hull [Nm]
    ballastT            // torque of keel about hull [Nm]
    dampT              // dampening torque [Nm]
    hullAacc            // angular acceleration of hull [rad/s^2]
oracle
    driving_force
    driving_dir
    sheet
    t                  // time [s]
derivate
    drag = dragK * hull_speed
    forward_force = driving_force * -sin(sheet) * driving_dir
    acceleration = (forward_force - drag) / boat_mass
    hull_speed = integ_wrt(acceleration, t)
    side_force = driving_force * cos(sheet) * driving_dir
    sailTq = side_force * mast_len / 3.0
    ballastT = ballast_weight * sin(list) * keel_depth * -driving_dir
    dampT = dampK * -hullAvel
    hullAacc = (sailTq + ballastT + dampT) / hull_moi
    hullAvel = integ_wrt(hullAacc, t)
    list = integ_wrt(hullAvel, t)
}

```

I generated the EDEN script for the rig and hull agents by transforming the derivates into definitions except for the integral derivates. The hull speed integral derivate, for example, was implemented by the EDEN action

```

/* hull_speed = integ_wrt(acceleration, t) */
proc integ_hull_speed : iClock {
    hull_speed = hull_speed_iVal + (acceleration * iPeriod / 2.0);
    hull_speed_iVal = hull_speed + (acceleration * iPeriod / 2.0);
}

```

and the EDEN action for the clock.

---

## A.5 Exploring the sailboat simulation

The SBS was constructed with the aim of recreating the experience of sailing so that I could use my knowledge of sailing directly to test theories and discover values for constants. The animation that allowed this use of knowledge is shown in Examples A.4 and A.5. The sailboat was represented by a view from above and from the stern (rear) and an interface was defined through which I was able to control the boat by turning it anticlockwise (port) or clockwise (starboard) and reducing the length of the sheet (sheeting-in) or increasing the length of the sheet (sheeting-out). A speed indicator gave the current speed of the boat. Later a clock and driving force indicator were added. Although primitive, the animation served its purpose of recreating the experience of sailing for testing the model and finding appropriate values for constants.

I was able to explore the SBS during its construction more as a sailor than a system developer. By interacting with the model via the interface or directly by changing the values of EDEN variables I explored the SBS:

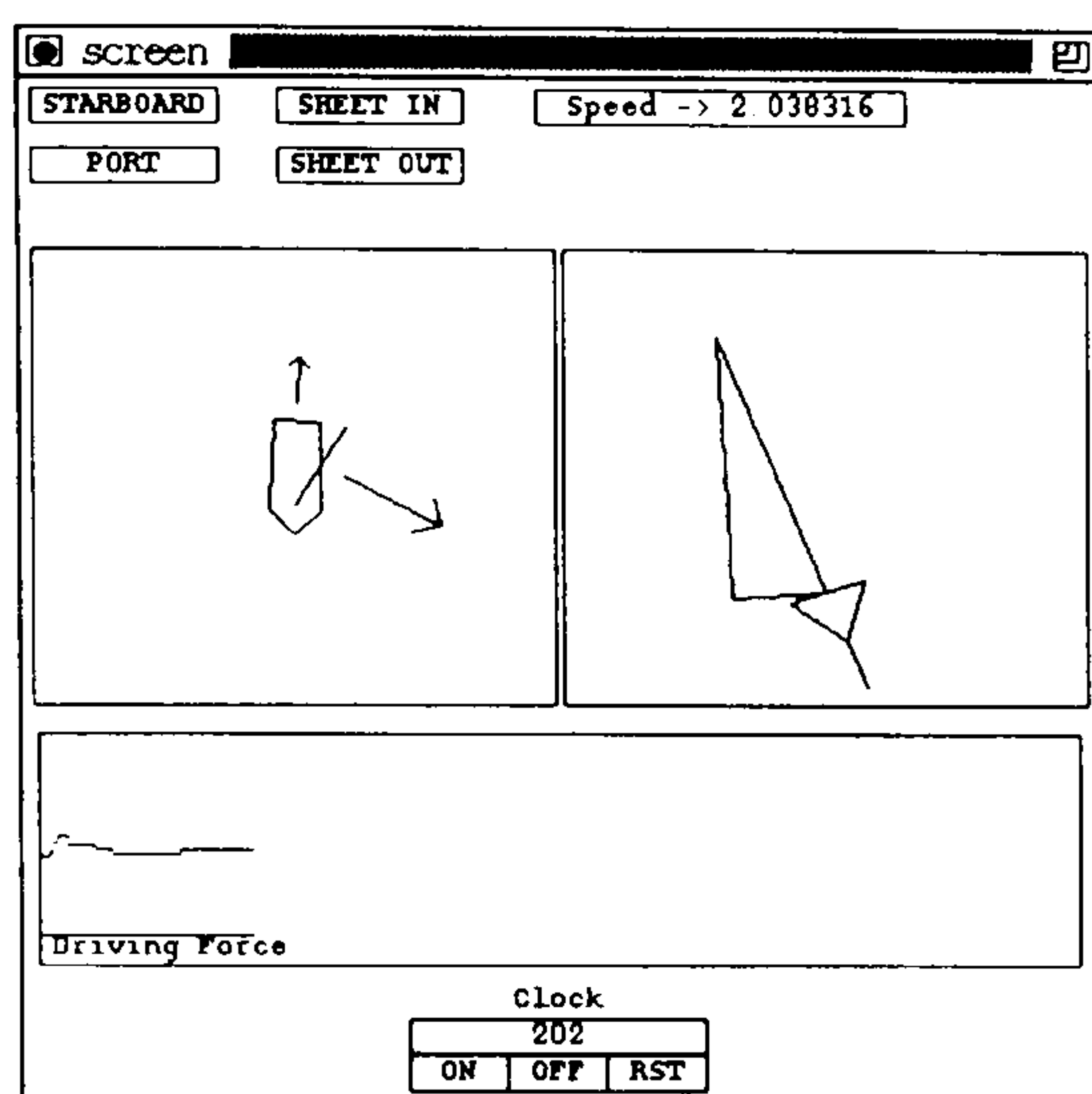
- searching for emergent behaviours;
- considering the behaviour in terms of physical principles;
- performing both familiar and novel manoeuvres;
- shifting the context of the sailing experience by changing variable values;
- looking for confirmation that the model is faithful to my experience;
- searching for behaviours which are not faithful to my experience.

Example A.4 shows four screen-shots while I was performing a common sailing manoeuvre called a “tack” in which the boat turns half-circle through the wind. Because of my familiarity with this common sailing manoeuvre I knew what to expect. If the model did not meet my expectations I either changed my beliefs, thus learning from my interaction with the simulation, or changed the model. For example, I discovered during interaction with the model that it was possible to

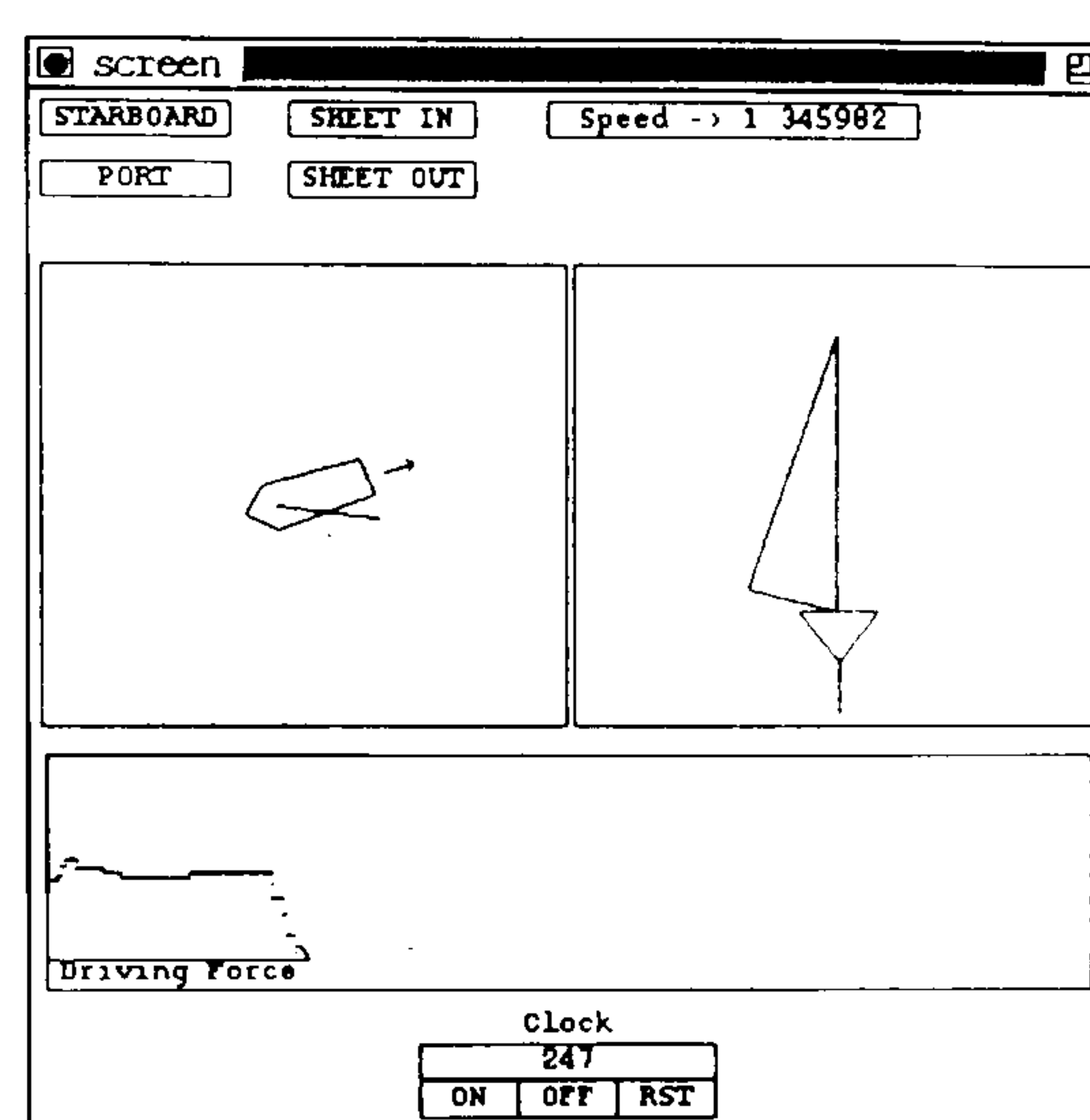


capsize the boat, as shown in Example A.5. This represented a change in my beliefs about the scope of the model.

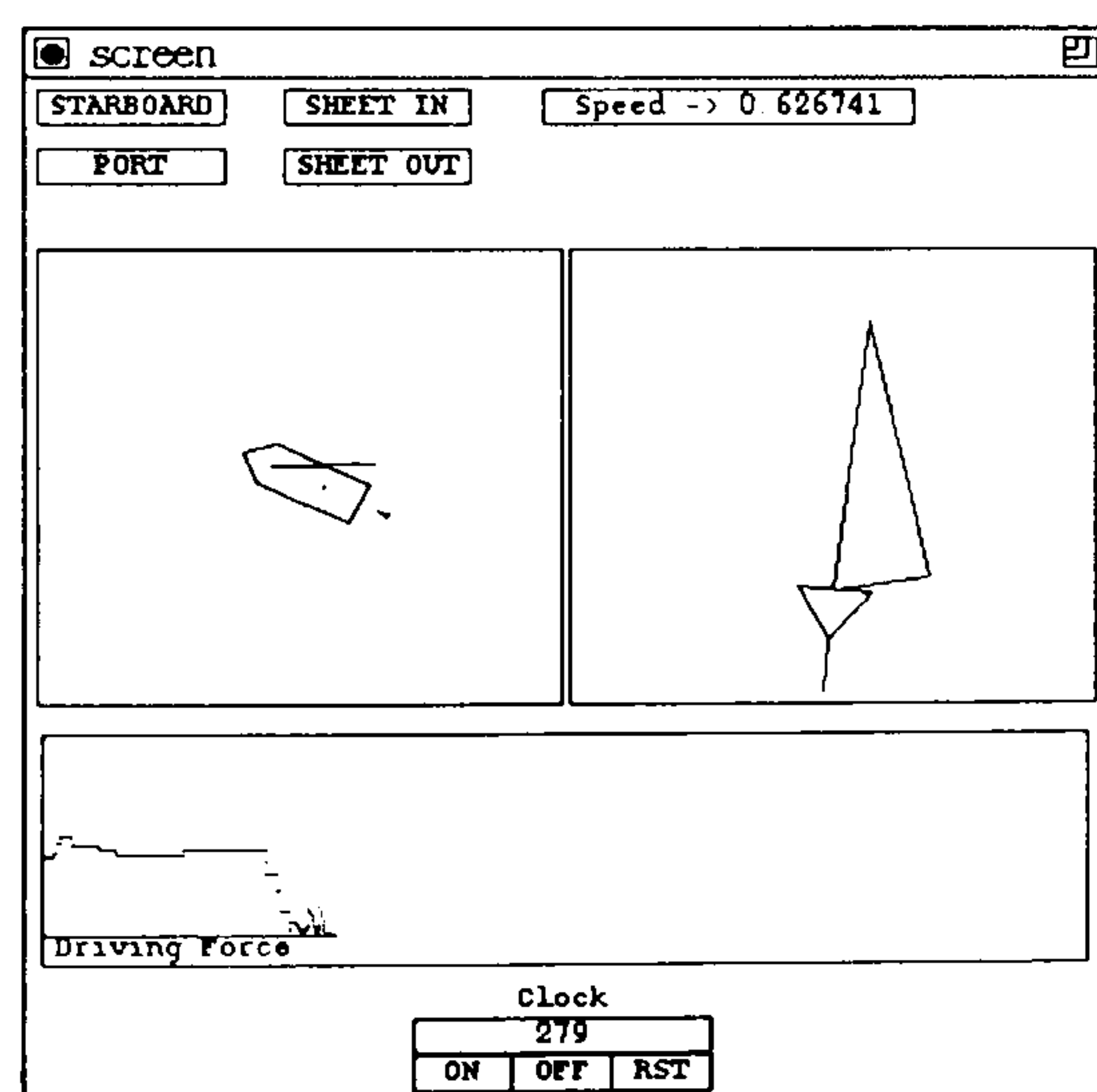
**Example A.4. Exploration in the SBS.** One way I explored the SBS was to perform familiar sailing manoeuvres. The following diagrams are screen-shots during a manoeuvre called a tack in which the sailboat turns half-circle through the wind.



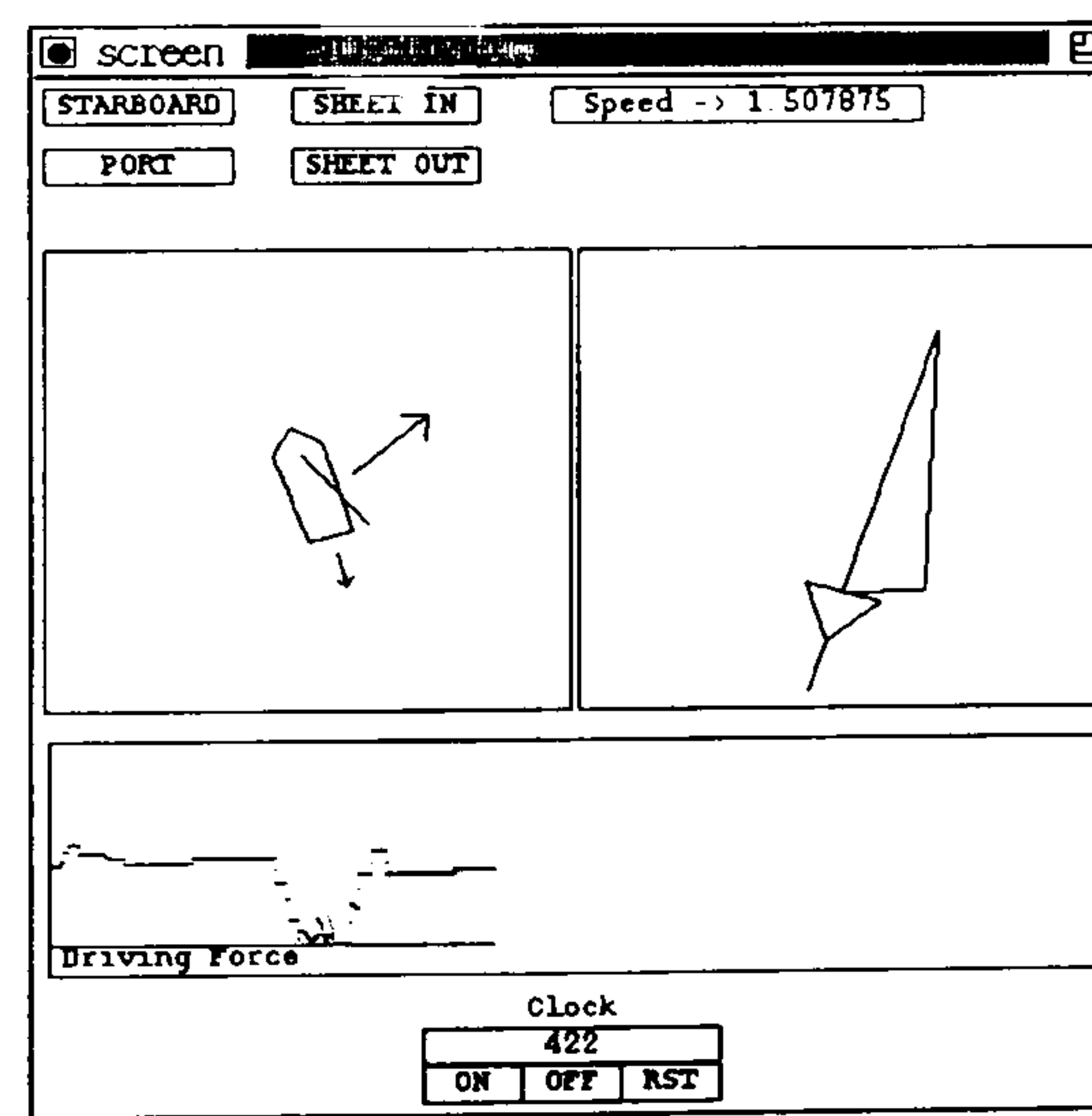
(a) Preparing to tack.



(b) Turning into wind (luffing).



(c) Sail changes sides.



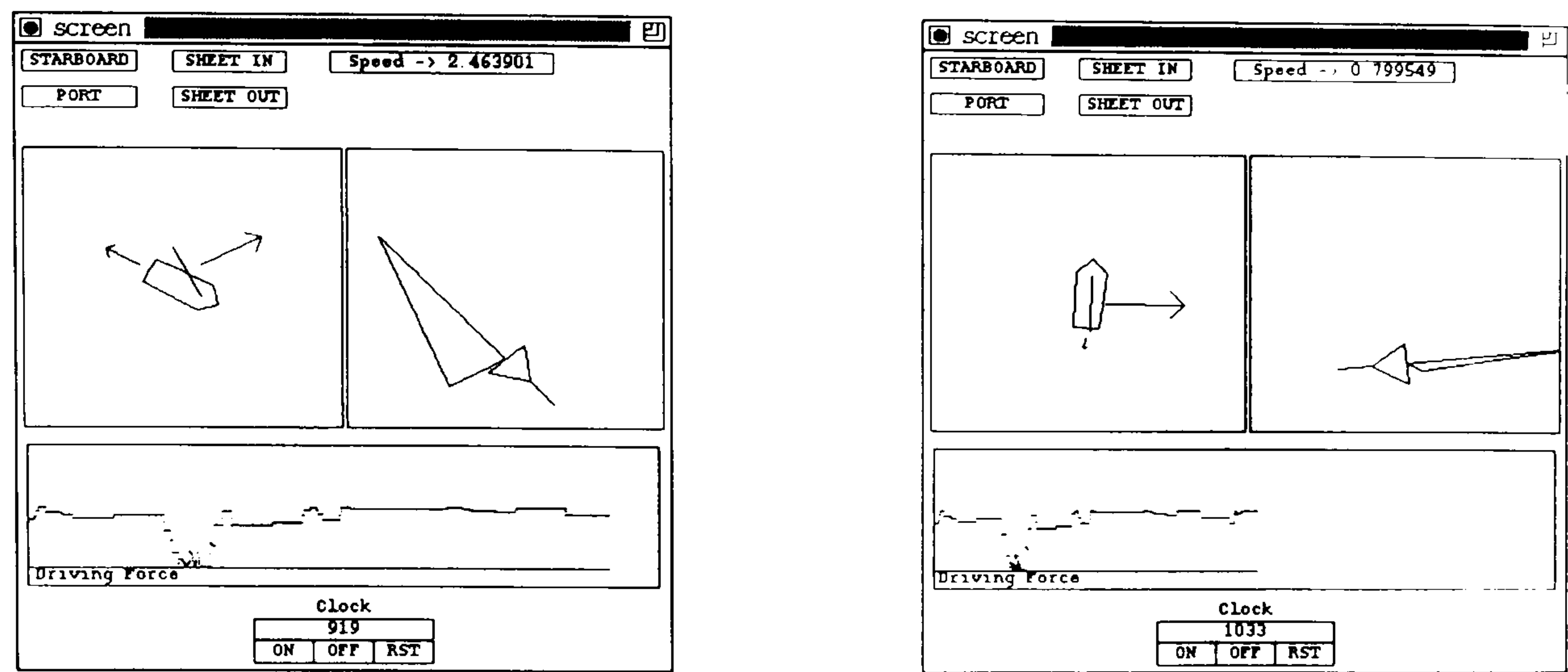
(d) Tack completed.

I found that the SBS provided a good approximation to the handling of a real sailboat.

## A.6 Extending the sailboat model

Once I was satisfied with the SBS I placed it in the Empirical Modelling Group archives. Later, another modeller retrieved it and continued to extend the model. This modeller had no experience of sailing but had discovered a book that detailed

**Example A.5. Emergence in the SBS.** It emerged during the exploration of the SBS that it represented the sailboat capsizing. This was a surprise to me because I did not think of the capsize situation during modelling.



(a) Turning to port.

(b) Sailboat capsizes.

This unexpected behaviour was discovered when I performed a manoeuvre incorrectly. The manoeuvre is called a “jibe” in which the stern (rear) passes through the wind thus making the sailboat unstable.

the phenomenon of turbulence on sails and the apparent shift in wind direction caused by the motion of the boat. The modeller was able to add this information to the existing LSD specification of the sail, as shown in Example A.6, without having to consult me. This provides evidence of the openness of models in EM and the ease with which they can be extended.



---

**Example A.6. Openness in the SBS.** Yung extended the LSD definition for the sail agent, shown in Example A.2, to include the effects of turbulence and the wind generated by the motion of the sailboat.

```

agent sail {
  const
    FpushK = 10 // pushing force constant [N m-3 s-2]
    FsucK = 20 // suction force constant [N m-3 s-2]
  state
    sail_dir      // sail direction as bearing [rad]
    driving_dir    // sail driven anti/clockwise [1/-1]
    driving_force  // driving force of sail [N]
    sheet_angle    // angle between keel and sail [rad]
    sail_area      // effective sail area [m2]
  oracle
    rel_wind_dir // wind direction experienced by the sail [rad]
    rel_wind_vel // wind speed experienced by the sail [m s-2]
    heading
    wind_dir
    sailAvel
  derivate
    sail_dir = heading + sheet
    sail_wind = rel_wind_dir - sail_dir
    driving_dir = (sin(sail_wind) < 0.0) ? 1 : -1
    driving_force = rel_wind_vel * abs(cos(sail_wind)) * FsucK * sail_area
                  + rel_wind_vel * abs(sin(sail_wind)) * FpushK * sail_area
    sail_area = boom_len * mast_len * cos(list)
    rel_wind_vel = sqrt(wind_vel2 + hull_speed2 -
                       2*wind_vel*hull_speed*cos(wind_dir - heading))
    rel_wind_dir =
      heading - asin(sin(wind_dir-heading) * wind_vel / rel_wind_vel) + pi
}

```

The new definition is essentially the the original with the the addition of more constants, observables and derivates.

---

## Appendix B

# Experiences Using the Shlaer-Mellor Method

This is a report prepared after an interview, by the author in November 1993, with those at IBM WSDL who were considering the prospect of adopting the Shlaer-Mellor object-oriented analysis and design method.

The Shlaer-Mellor object-oriented analysis and design method [SM88, SM92] is currently being investigated at the IBM WSDL (Warwick Software Development Laboratory) [DSWW93] and other IBM sites as a means of improving their software quality and productivity. The method is being used in a project to improve a report generator. This main project has spawned two mini-projects. The first project followed the Shlaer-Mellor approach to produce inefficient yet working C code. The second project is still in progress, aiming to build on the work of the previous project by reusing the analysis work done, using a multitasking software architecture and automatically generating C++ code.

The Shlaer-Mellor method is being considered at the IBM WSDL as a replacement for the traditional in-house SD approach based on the use of work-books. The work-books are used by software developers to specify the software for a system, then passed to programmers who code the software in C or the IBM Application System Language (ASL). Programming in ASL involves composing application modules



in accordance with their Application Program Interfaces (API). The existing version of the report generator was developed using the in-house approach and written in ASL.

The sections that follow the overview of the Shlaer-Mellor method give an account of what was learned about software development using the Shlaer-Mellor method in the IBM WSDL through talking with those actively involved at both the technical and managerial levels.

## B.1 Overview of the Shlaer-Mellor method

Shlaer and Mellor [SM88, SM92] developed their object-oriented analysis method over the course of several years of consulting practice in information modelling [Mar90]. Although information modelling forms the foundation of the method it also draws from conventional object-oriented analysis methods that model the behaviour and function of systems. The Shlaer-Mellor object-oriented analysis method has the following sequence of stages:

1. Large problems are decomposed into conceptually distinct domains. Four main types of domains are identified in the method: application, service, architectural and implementation domains. Bridges link domains together.
2. The software developer follows domain analysis by constructing an information model or entity-relationship diagram (ERD). The information model consists of objects classes and their attributes with inheritance and aggregation relations defined between them.
3. The software developer defines lifecycles for the objects and relations as state models consisting of states, events, transitions and actions. During this stage the software developer defines timers and other mechanisms for managing concurrent behaviour.
4. An action data-flow diagram (ADFD) is defined for each action in the state model. Actions consist of four types of process: data transformation, data access, data testing and event generation.

In their second book [SM92] Shlaer and Mellor extend their method from the problem domain into the solution domain by describing a transformation from the products of object-oriented analysis to the products of object-oriented design.

## B.2 Domain analysis

The following was discovered about domain analysis by talking to those responsible for constructing the end-user-interface for the new application:

- It is possible to determine the client-server relations between modules in the existing system by analyzing their interface definitions and the configuration of modules.
- The service domain consists of all those modules which provide a service in the existing system and the application domain consists of all those modules which are clients in the existing system.
- Client-server relations between modules in the existing system map onto bridges between the application and service domains and bridges between subsystems within domains.
- It is difficult to define bridges to the implementation domain because it is a domain that has yet to be defined. The IBM WSDL is investigating ways of defining a “wrapper” interface for domains which have yet to be defined based on APIs.

These findings suggest that the Shlaer-Mellor method of domain analysis is suitable so long as the system being analyzed is already divided into parts that correspond to domains linked by bridges. In the absence of such a correspondence, domain analysis becomes difficult. This is exacerbated when domains have yet to be defined.

## B.3 Using the Teamwork tool in analysis and design

The following was discovered about the use of the Teamwork computer-aided software engineering tool for analysis and design by talking to the various people who



had used the tool:

- The tool is configured to support the notations and principles of the Shlaer-Mellor method. The tool can also be configured for other object-oriented analysis and design methods.
- The most useful feature of the tool was found to be the repository of the products of analysis and design that can be accessed across a network by software developers and provides the source for automatic code generation (Section B.4).
- Developers mainly use the tool for drawing diagrams. The other facilities were found of little use and even as a drawing tool it has its limitations with no intelligent text entry or diagram reformatting.
- The representations of models tend to be much larger than the screen. Developers tend to print parts of the model onto pieces of paper and stick them together to see more than the screen can show.

The above findings suggest that the ability of the tools to hold and distribute information was more useful to the software developers than its ability to process or represent information. A more generalized tool or suite of tools that supports access to a knowledge base and diagramming may have been more appropriate.

## B.4 Automatic code generation

The following was discovered about code generation by talking to the person responsible for building the system that performed the automatic conversion of models into code:

- The software architecture for the new system is defined as object class definitions which are reusable. The object classes include the “engine” that drives the state machine and generalized classes for the states, transitions, etc.
- Code is automatically generated by the tool in the following sequence of steps:

1. The information and state models constructed during analysis are retrieved from the Teamwork repository (Section B.3).
2. A skeleton object class definition is generated for each application object in the information model consisting of class, attribute and action names.
3. Specialized state, event and transition object class definitions are generated for each object state model.

Programmers complete the skeleton object class definitions by writing code to implement the actions.

- The generated code consists of an object class definition for each application object, state, event and transition represented during analysis resulting in a large number of definitions.

Automatic generation of code was a feature that attracted IBM WSDL to the Shlaer-Mellor method. However, the findings suggest the generated code is unmanageable because of its complexity. This makes it difficult to test and maintain code.

## B.5 Testing

The following was discovered about testing by talking to the various people who were responsible for devising means for checking the quality of code:

- Testing is performed on the products of analysis because they tend to be more structured than the code.
- Tools are required to help understand the complex behaviour resulting from the simulation of multiple state machines acting concurrently. The decision was made not to use the IBM Tuscon Object Oriented Analysis Simulator (TOOAS). Instead, a role-based method is being used at the IBM WSDL in which objects are isolated in turn and their role in the behaviour of the system examined.
- The tools help the tester to visualize the dynamics described in the model and it automatically checks that constraints defined by the tester are being satisfied.



These findings suggest that products of analysis are difficult to understand without the help of powerful simulation tools. They also suggest that the products of analysis define the same behaviour as the generated code. This is probably due to the automatic code generation.

## B.6 Conclusion

The Shlaer-Mellor method is far more prescriptive than the traditional approach of using work-books. This has the advantage of tool support and powerful techniques for analysis. However, some of the features of tools and techniques were found by software developers to be too limited. Developers found that the method could only be applied when there was a direct correspondence between the system being analyzed and the notations and principles of the method. Some software developers voiced their reservations about the method and its lack of support for using their experience and knowledge of developing software.

## Appendix C

# SUL, MUL and Hydrolift Artefacts

### C.1 SUL artefacts

#### C.1.1 SUL LSD specification

```
agent door() {
state
  door
oracle
  brake
derivate
  door is (brake == ON) ? OPEN : CLOSED
}

agent landing(_F) {
state
  landButton
oracle
  floor direction brake
handle
  brake destination
protocol
  landButton[_F] == ON && _F == floor + direction && brake == OFF -> brake = ON,
  landButton[_F] == ON && direction == NIL -> destination = _F,
  floor == _F -> landButton[_F] = OFF
}

agent car(_F) {
state
  carButton
oracle
  floor direction brake
handle
  brake destination
protocol
  carButton[_F] == ON && _F == floor + direction && brake == OFF -> brake = ON,
  carButton[_F] == ON && direction == NIL -> destination = _F,
  floor == _F -> carButton[_F] = OFF
}
```

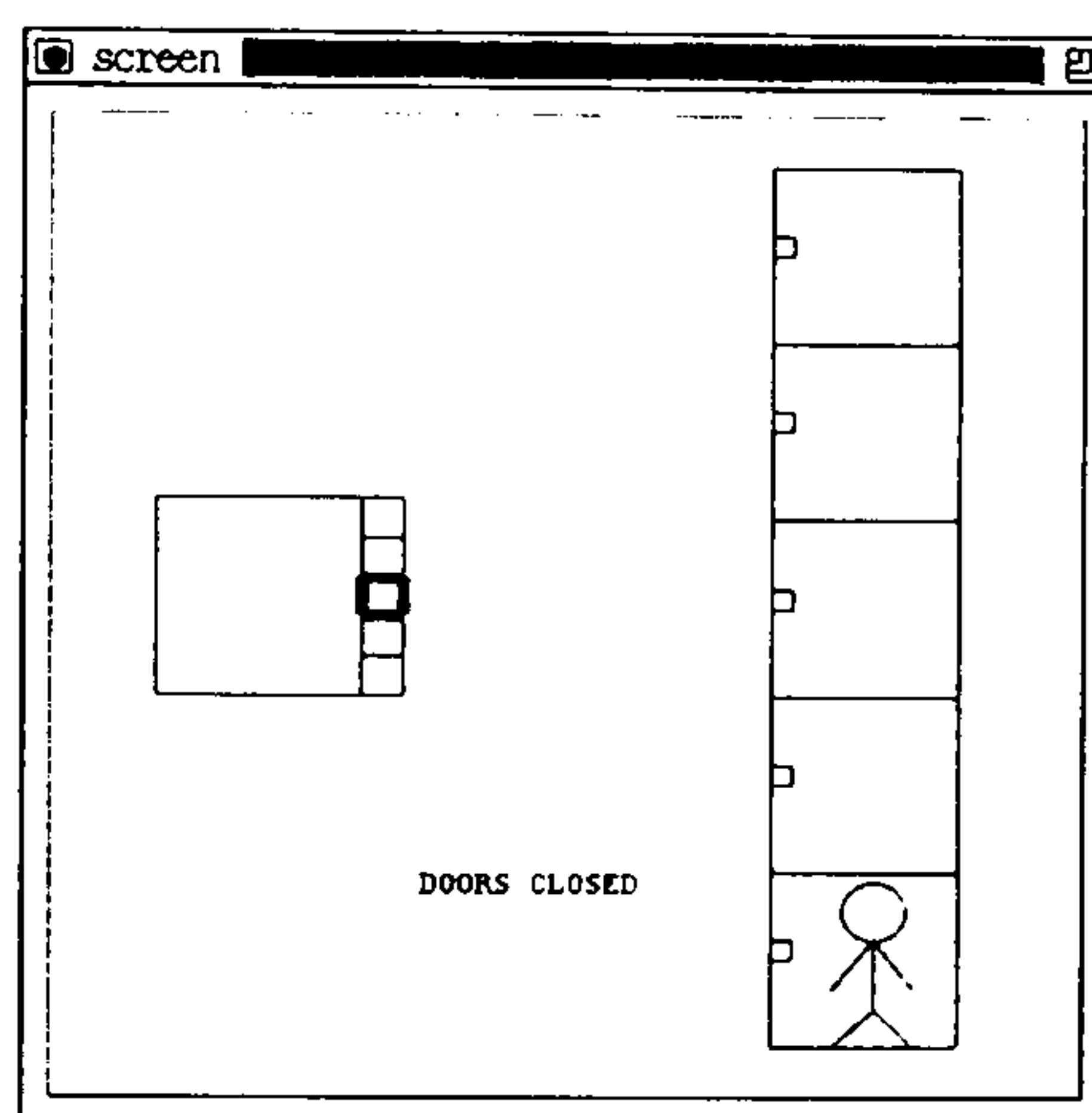


```

agent shaft() {
state
  floor destination direction
oracle
  brake
handle
  brake
derivate
  direction is (floor < destination) ? UP :
               (floor > destination) ? DOWN : NIL
protocol
  brake == OFF -> floor = floor + direction,
  brake == ON && direction != NIL -> brake = OFF
}

```

### C.1.2 SUL visualization/animation



### C.1.3 SUL DoNaLD script

The following DoNaLD script defines the SUL visualization.

```

%donald
#####
#LIFT USER#
#####

openshape man
within man {
  circle head
  line body, leftarm, rightarm, leftleg, rightleg
  point manpos
  int rad
  manpos = {0,0}
  rad = 15
  head = circle(manpos, rad)
  body = [manpos - {0,rad}, manpos - {0,50}]
  leftarm = [manpos - {0,rad}, manpos - {20,40}]
  rightarm = [manpos - {0,rad}, manpos - {-20,40}]
  leftleg = [manpos - {0,50}, manpos - {20,70}]
  rightleg = [manpos - {0,50}, manpos - {-20,70}]
}

boolean inlift
inlift = false

openshape person
within person {

```

```

shape person
point put, one
put = {~/carpos.1 + 100, ~/carpos.2 + 150}
one = {800, 10}
person = if ~/inlift then trans( scale(~/man, 2), put.1, put.2 ) else
      trans( scale(~/man, 2), one.1, one.2+(~/floor*180) )
}

#####
#LIFT CAR#
#####

openshape box
within box {
  int width, length
  point p, q, b, d
  line top, bot, left, right
  b = {0, 0}
  d = b + {width, 0}
  p = b + {0, length}
  q = b + {width, length}
  width, length = 100, 100
  top = [p, q]
  bot = [b, d]
  left = [p, b]
  right = [q, d]
}

int floor
floor = 1

point carpos
carpos = {100, 50+(180*(floor-1)) }

openshape car
within car {
  point corner
  corner = ~/carpos
  shape car
  car = trans( scale(~/box, 2), corner.1, corner.2)
  int X, Y
  X = 200
  real ratio
  ratio = 0.4

  openshape button1
  within button1 {
    shape button1
    boolean light
    light = true
    button1 = trans( scale(~/~/box, ~/ratio), ~/corner.1+~/X,~/corner.2)
  }

  openshape button2
  within button2 {
    shape button2
    button2 = trans( scale(~/~/box, ~/ratio), ~/corner.1+~/X, ~/corner.2+~/ratio*100)
    boolean light
    light = false
  }

  openshape button3
  within button3 {
    shape button3

```



```

    button3 = trans( scale(~/~/box, ~/ratio), ~/corner.1+~/X, ~/corner.2+~/ratio*200)
    boolean light
    light = false
}

openshape button4
within button4 {
    shape button4
    button4 = trans( scale(~/~/box, ~/ratio), ~/corner.1+~/X, ~/corner.2+~/ratio*300)
    boolean light
    light = false
}

openshape button5
within button5 {
    shape button5
    button5 = trans( scale(~/~/box, ~/ratio), ~/corner.1+~/X, ~/corner.2+~/ratio*400)
    boolean light
    light = false
}
}

? A_car_button1_button1 is (carButton_1 == ON) ? "linewidth=5" : "linewidth=0";
? A_car_button2_button2 is (carButton_2 == ON) ? "linewidth=5" : "linewidth=0";
? A_car_button3_button3 is (carButton_3 == ON) ? "linewidth=5" : "linewidth=0";
? A_car_button4_button4 is (carButton_4 == ON) ? "linewidth=5" : "linewidth=0";
? A_car_button5_button5 is (carButton_5 == ON) ? "linewidth=5" : "linewidth=0";

#####
#LANDINGS#
#####

int ceiling, wall
ceiling = 950
wall = 700
real ratio
ratio = 1.8

openshape floor1
within floor1 {
    shape floor1
    floor1 = trans( scale(~/box, ~/ratio), ~/wall, ~/ceiling-~/ratio*500)
    openshape button
    within button {
        shape button
        boolean light
        light = false
        button = trans( scale(~/~/box, 0.2), ~/~/wall, ~/~/ceiling-~/~/ratio*450)
    }
}

openshape floor2
within floor2 {
    shape floor2
    floor2 = trans( scale(~/box, ~/ratio), ~/wall, ~/ceiling-~/ratio*400)
    openshape button
    within button {
        shape button
        boolean light
        light = false
        button = trans( scale(~/~/box, 0.2), ~/~/wall, ~/~/ceiling-~/~/ratio*350)
    }
}
}

```

```

openshape floor3
within floor3 {
  shape floor3
  floor3 = trans( scale(~ /box, ~ /ratio), ~ /wall, ~ /ceiling-~ /ratio*300)
  openshape button
  within button {
    shape button
    boolean light
    light = false
    button = trans( scale(~ /~ /box, 0.2), ~ /~ /wall, ~ /~ /ceiling-~ /~ /ratio*250)
  }
}

openshape floor4
within floor4 {
  shape floor4
  floor4 = trans( scale(~ /box, ~ /ratio), ~ /wall, ~ /ceiling-~ /ratio*200)
  openshape button
  within button {
    shape button
    boolean light
    light = false
    button = trans( scale(~ /~ /box, 0.2), ~ /~ /wall, ~ /~ /ceiling-~ /~ /ratio*150)
  }
}

openshape floor5
within floor5 {
  shape floor5
  floor5 = trans( scale(~ /box, ~ /ratio), ~ /wall, ~ /ceiling-~ /ratio*100)
  openshape button
  within button {
    shape button
    boolean light
    light = false
    button = trans( scale(~ /~ /box, 0.2), ~ /~ /wall, ~ /~ /ceiling-~ /~ /ratio*50)
  }
}

? A_floor1_button_button is (landButton_1 == ON) ? "linewidth=5" : "linewidth=0";
? A_floor2_button_button is (landButton_2 == ON) ? "linewidth=5" : "linewidth=0";
? A_floor3_button_button is (landButton_3 == ON) ? "linewidth=5" : "linewidth=0";
? A_floor4_button_button is (landButton_4 == ON) ? "linewidth=5" : "linewidth=0";
? A_floor5_button_button is (landButton_5 == ON) ? "linewidth=5" : "linewidth=0";

```

#### C.1.4 SUL ADM script

The following script defines the ADM entities for the SUL animation.

```

%adm
entity door() {
  definition
    door is (brake == ON) ? OPEN : CLOSED
  }

  entity landing(_F) {
    action
      landButton{_F} == ON && _F == floor + direction && brake == OFF -> brake = ON,
      landButton{_F} == ON && direction == NIL -> destination = _F,
      floor == _F -> landButton{_F} = OFF
    }
  }

```



```

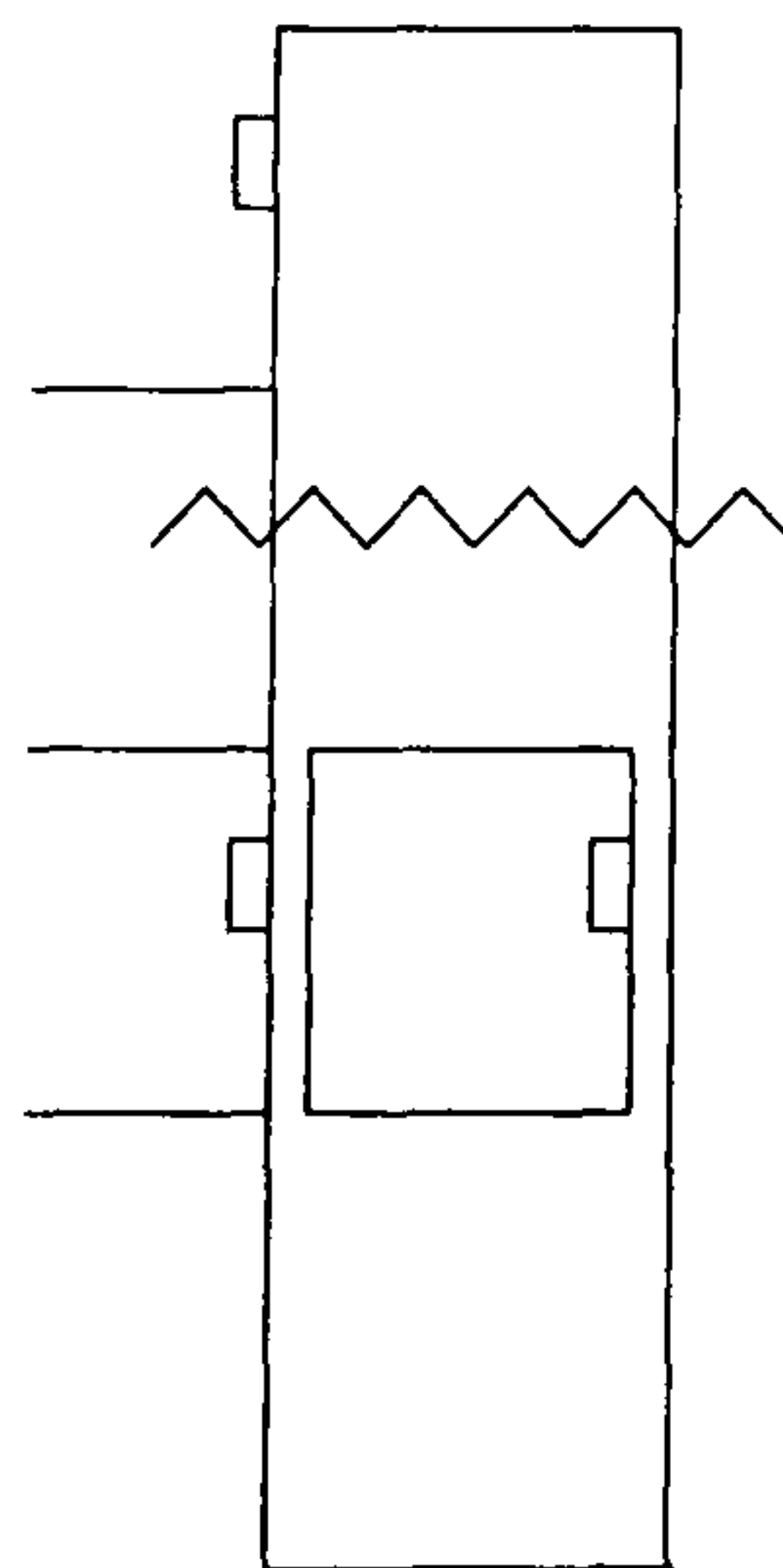
entity car(_F) {
  action
    carButton[_F] == ON && _F == floor + direction && brake == OFF -> brake = ON,
    carButton[_F] == ON && direction == NIL -> destination = _F,
    floor == _F -> carButton[_F] = OFF
}

entity shaft() {
  definition
    direction is (floor < destination) ? UP :
                  (floor > destination) ? DOWN : NIL
  action
    brake == OFF -> floor = floor + direction,
    brake == ON && direction != NIL -> brake = OFF
}

# instantiate new entities
door()
shaft()
car(1)
car(2)
car(3)
car(4)
car(5)
landing(1)
landing(2)
landing(3)
landing(4)
landing(5)

```

### C.1.5 SUL sketch



### C.1.6 SUL statement of requirements

On each landing there is a button and in the car there is a button for each floor. The user makes a request for the car to come to his landing by pressing a button. The shaft mechanism moves the car to his landing and opens the door. The user enters the car and presses a button. The shaft mechanism moves the car to the landing he requested and opens the door. The user exits the car. For safety the door is opened and

closed by the brake ensuring that the door is only open whilst the brake is on.

## C.2 MUL artefacts

### C.2.1 MUL LSD specification

```

agent door() {
state
  door
oracle
  brake
derivate
  door is (brake == ON) ? OPEN : CLOSED
}

agent landing(_F) {
state
  landButton
oracle
  floor direction brake
handle
  brake destination
protocol
  landButton[_F] == UP && _F == floor + 1 && brake == OFF -> brake = ON,
  landButton[_F] == DOWN && _F == floor - 1 && brake == OFF -> brake = ON,
  landButton[_F] != OFF && direction == NIL -> destination = _F,
  floor == _F -> landButton[_F] = OFF
}

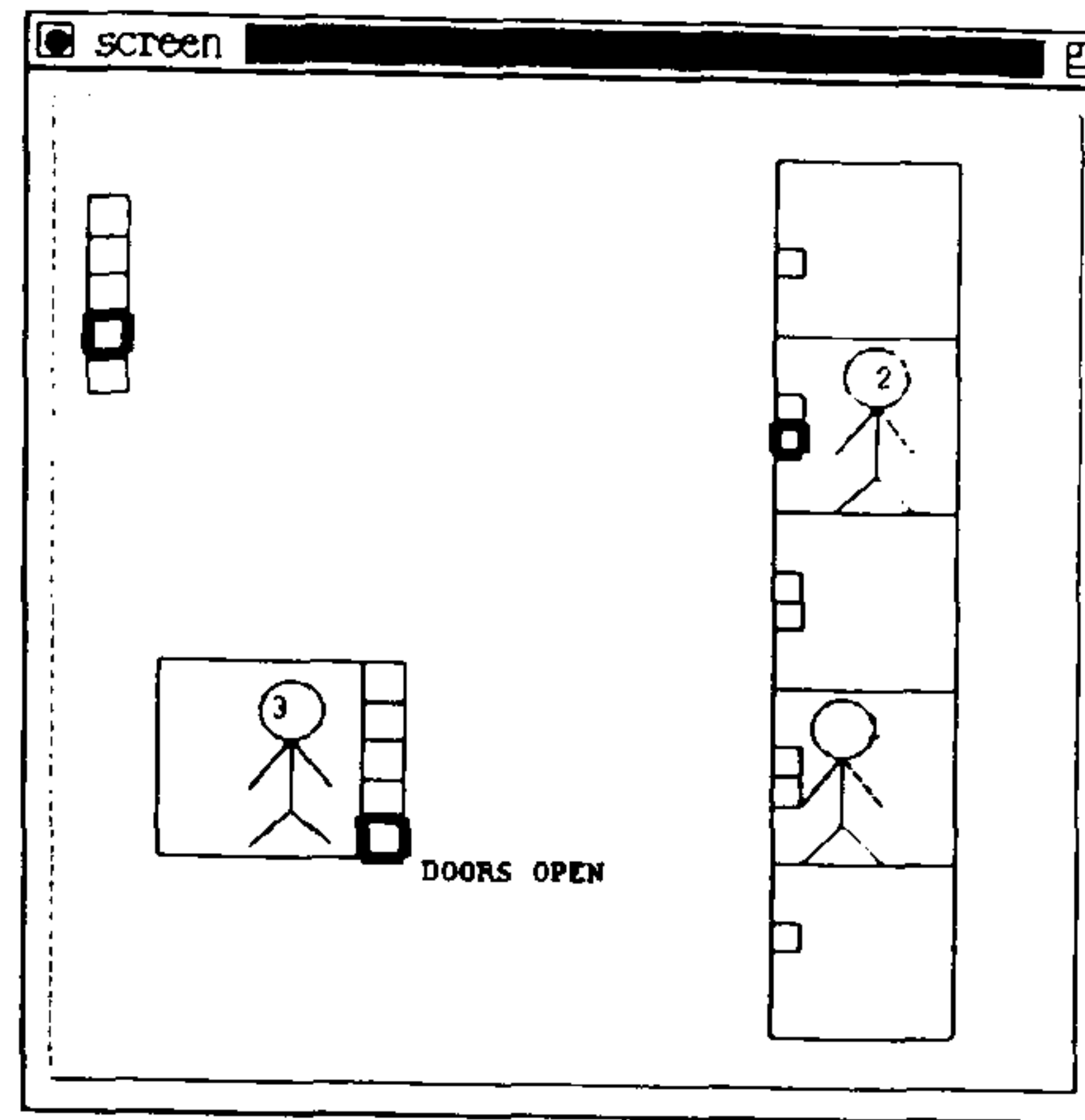
agent car(_F) {
state
  carButton
oracle
  floor direction brake
handle
  brake destination
protocol
  carButton[_F] == ON && _F == floor + direction && brake == OFF -> brake = ON,
  carButton[_F] == ON && direction == NIL -> destination = _F,
  floor == _F -> carButton[_F] = OFF
}

agent shaft() {
state
  floor destination direction
oracle
  brake
handle
  brake
derivate
  direction is (floor < destination) ? UP :
               (floor > destination) ? DOWN : NIL
protocol
  brake == OFF -> floor = floor + direction,
  brake == ON && direction != NIL -> brake = OFF
}

```



### C.2.2 MUL visualization/animation



### C.2.3 MUL DoNaLD redefinitions

The following DoNaLD script redefines the SUL visualization as the MUL visualization by redefining the person shape to represent three people.

```
%donald
#####
#LIFT USERS#
#####

boolean inlift, inliftB, inliftC
inlift = false
inliftB = false
inliftC = false

openshape person
within person {
  shape person, person2, person3
  point put, one
  label p1, p2, p3
  p1 = if ~/inlift then label("1", put) else
        label("1", {one.1,one.2+(~/floor*180)})
  p2 = if ~/inliftB then label(" 2", put) else
        label(" 2", {one.1,one.2+(~/floorB*180)})
  p3 = if ~/inliftC then label("  3", put) else
        label("  3", {one.1,one.2+(~/floorC*180)})
  put = {~/carpos.1 + 100, ~/carpos.2 + 150}
  one = {800, 10}
  person = if ~/inlift then trans( scale(~/man, 2), put.1-30, put.2 ) else
            trans( scale(~/man, 2), one.1-30, one.2+(~/floor*180) )
  person2 = if ~/inliftB then trans( scale(~/man, 2), put.1, put.2 ) else
            trans( scale(~/man, 2), one.1, one.2+(~/floorB*180) )
  person3 = if ~/inliftC then trans( scale(~/man, 2), put.1+30, put.2 ) else
            trans( scale(~/man, 2), one.1+30, one.2+(~/floorC*180) )
}
```

### C.2.4 MUL ADM redefinitions

The following ADM script redefines entities in the SUL animation by instantiating new user entity definitions and redefining the landing entity.

```
%adm
entity userIn(_U) {
```

```

definition
  floor{_U} is floor
action
  Rand(2) == 1 && door == OPEN -> delete_userIn(_U); floor{_U} = floor; userOut(_U),
  Rand(2) == 1 -> execute("carButton_//str(Rand(5))//\" = ON;")
}

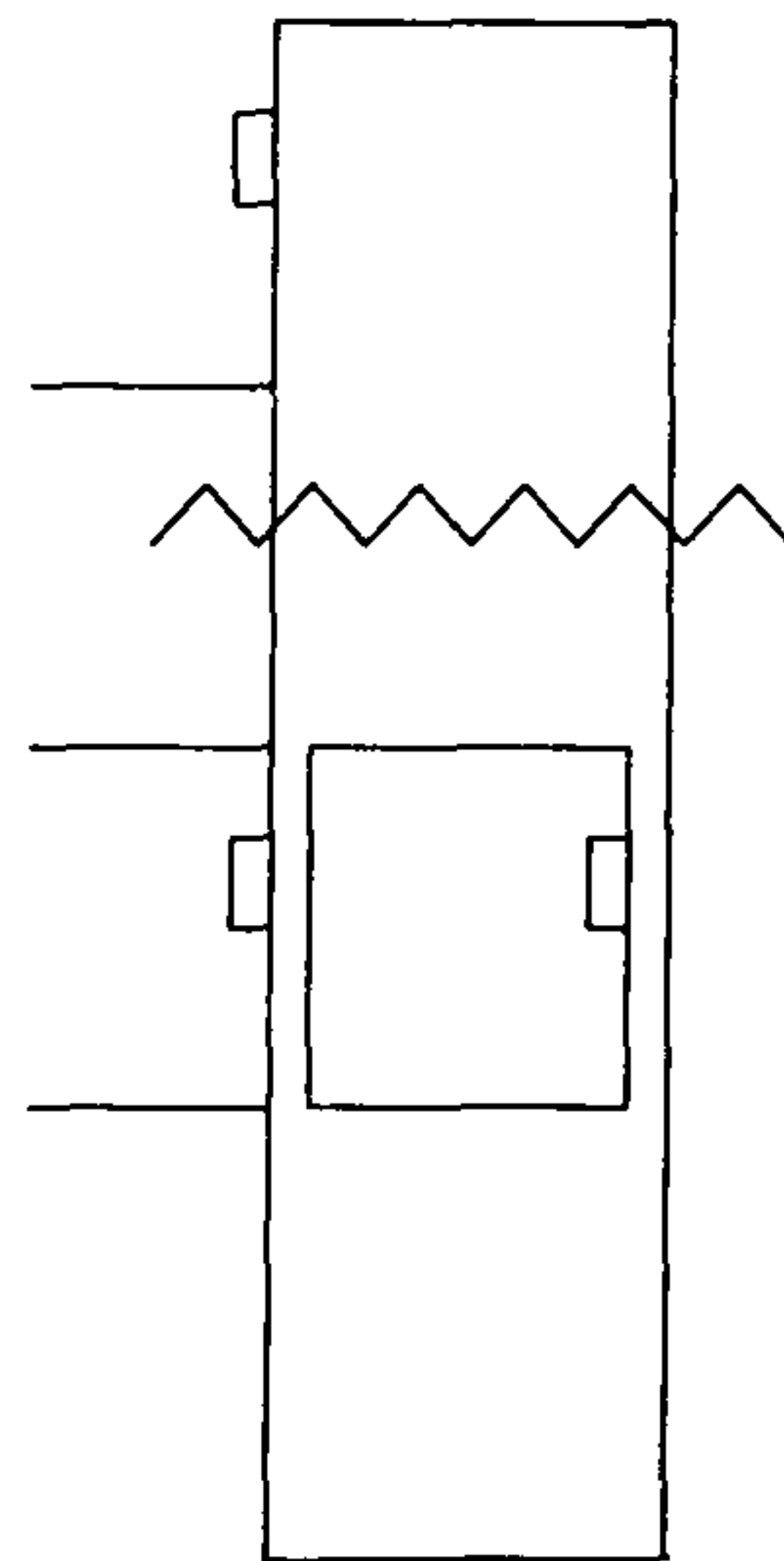
entity userOut(_U) {
action
  Rand(2) == 1 && door == OPEN && floor == floor{_U} -> delete_userOut(_U); userIn(_U),
  Rand(2) == 1 -> execute("landButton_//str(floor{_U})//\" = UP;"),
  Rand(2) == 1 -> execute("landButton_//str(floor{_U})//\" = DOWN;")
}

entity landing(_F) {
action
  landButton{_F} == UP && _F == floor + 1 && brake == OFF -> brake = ON,
  landButton{_F} == DOWN && _F == floor - 1 && brake == OFF -> brake = ON,
  landButton{_F} != OFF && direction == NIL -> destination = _F,
  floor == _F -> landButton{_F} = OFF
}

# instantiate new entities
userIn(1)
userIn(2)
userIn(3)

```

### C.2.5 MUL sketch

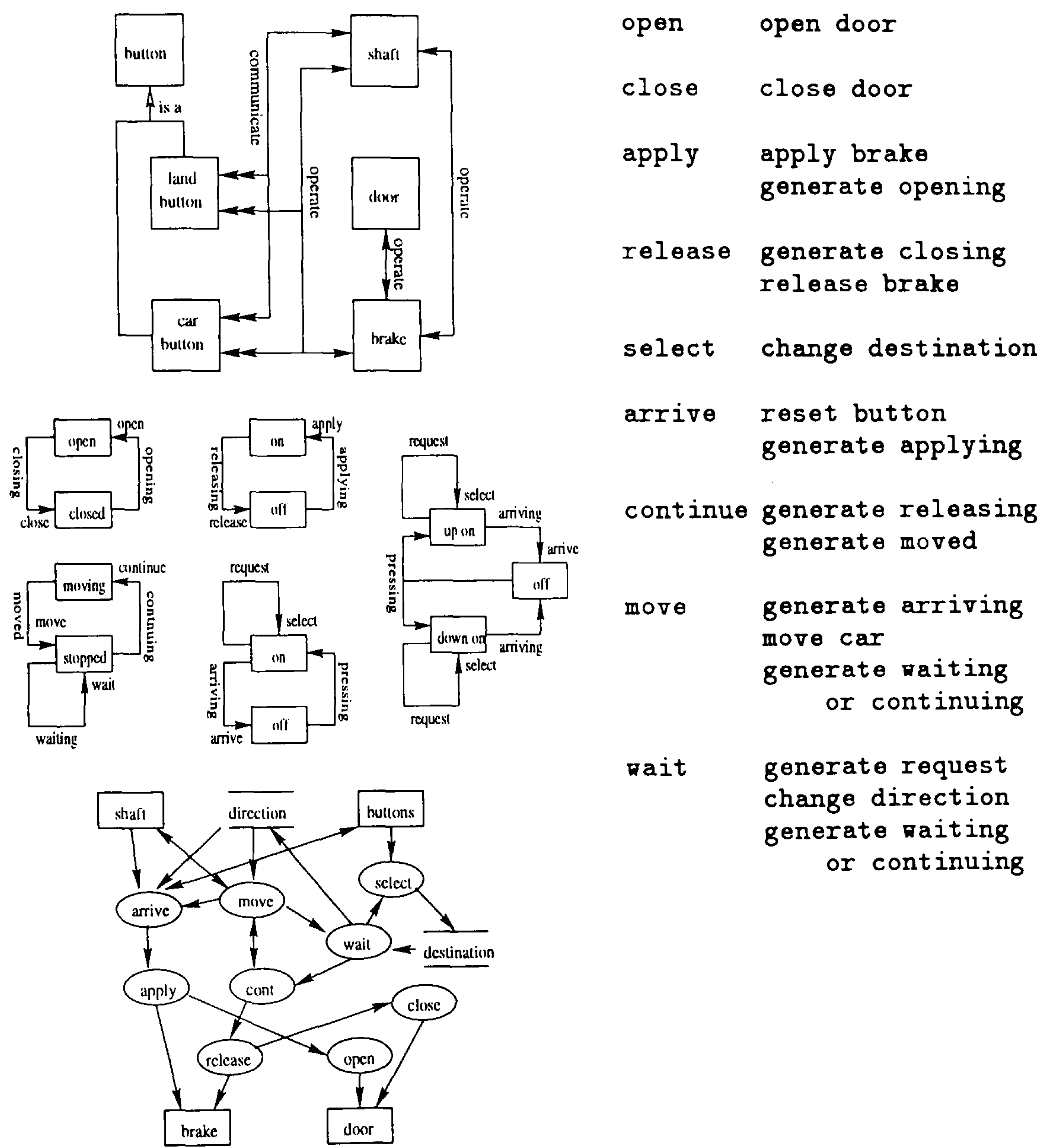


### C.2.6 MUL statement of requirements and models

On each landing there is an up and a down button. In the car there is a button for each floor. Users make requests for the car to come to their landing or go to another landing by pressing these buttons. The shaft mechanism moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user (for requests from landings the direction matters). On arriving at the destination landing the shaft mechanism selects the next destination. The shaft mechanism releases



the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.



- open      open door
- close     close door
- apply     apply brake  
          generate opening
- release   generate closing  
          release brake
- select    change destination
- arrive    reset button  
          generate applying
- continue   generate releasing  
          generate moved
- move      generate arriving  
          move car  
          generate waiting  
          or continuing
- wait      generate request  
          change direction  
          generate waiting  
          or continuing

### C.3 Hydrolift artefacts

#### C.3.1 Hydrolift LSD specification

```
agent door() {
state
  door
oracle
  brake
derivate
  door is (brake == ON) ? OPEN : CLOSED
}

agent landing(_F) {
state
  landButton
oracle
  sensed brake
handle
```

```

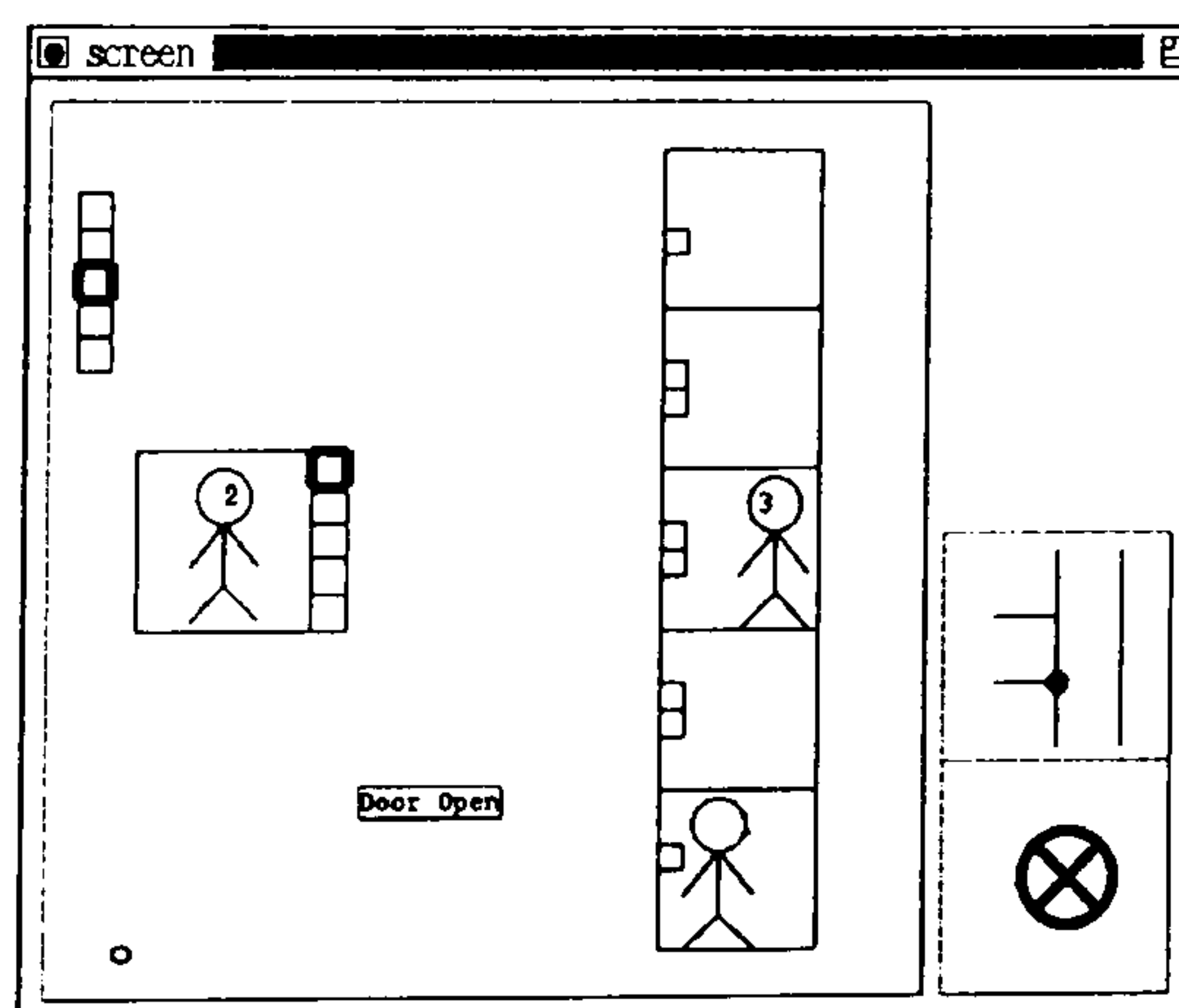
    brake
protocol
    landButton{_F} == UP && sensed{_F - 1} == UP && brake == OFF -> brake = ON,
    landButton{_F} == DOWN && sensed{_F + 1} == DOWN && brake == OFF -> brake = ON,
    sensed{_F} != NIL -> landButton{_F} = OFF
}

agent car(_F) {
    state
        carButton
    oracle
        chan2
    handle
        chan1
    protocol
        carButton{_F} == ON -> chan1 = _F,
        chan2 == _F -> carButton{_F} = OFF
}

agent pump() {
    state
        change target
    oracle
        brake pressure chan1
    handle
        brake pressure chan2
    derivate
        k = 100,
        change is (pressure < target) ? k :
                  (pressure > target) ? -k : 0
    protocol
        target == pressure + change && brake == OFF -> brake = ON,
        change == 0 -> target = chan1*k,
        pressure == target -> chan2 = target/k,
        brake == OFF -> pressure = pressure + change,
        brake == ON && change != 0 -> brake = OFF
}

```

### C.3.2 Hydrolift visualization/animation



### C.3.3 Hydrolift DoNaLD redefinitions

The following DoNaLD script redefines the MUL visualization as the Hydrolift visualization by adding shapes for the pump and valve.

```

#####
#PUMP#

```



#####

```

openshape pumpshape
within pumpshape {
  circle base
  point pos
  int radius
  boolean on
  base = circle(pos, radius)
  pos = {460,350}
  radius = 40
  on = false
  line one, two
  point p1, p2, p3, p4
  int const
  one = [ p1 , p2 ]
  two = [ p3 , p4 ]
  const = 29
  p1 = pos - {const, const}
  p2 = pos + {const, const}
  p3 = pos + {-const, const}
  p4 = pos + {const, -const}
}

```

```

? A_pumpshape_one is (change > 0) ? "linewidth=5" : "linewidth=0";
? A_pumpshape_two is (change > 0) ? "linewidth=5" : "linewidth=0";
? A_pumpshape_base is (change > 0) ? "linewidth=5" : "linewidth=0";

```

```

#####
#VALVE#
#####

```

```

openshape diaphragm
within diaphragm {
  line pipe1, pipe2, pipe3, pipe4, pipe5
  point one, two, three, four, five, six, seven, eight
  int height, width
  point pos
  height = 300
  width = 100
  pos = {200, 200}
  two = pos + {0, height}
  three = pos + {-width, 0}
  four = pos + {-width, 100}
  pipe1 = [ pos, two]
  pipe2 = [ three, four ]
  five = four + {0, 100}
  six = three + {0, height}
  pipe3 = [ five, six ]
  seven = five - {100, 0}
  eight = four - {100, 0}
  pipe4 = [ seven, five ]
  pipe5 = [ eight, four ]
}

```

```

openshape valve
within valve {
  circle base
  int radius
  base = circle(~/diaphragm/four, radius)
  radius = 10
  line valvepos
  point end
  int change

```

```

boolean open
change = if open then ~/diaphragm/width else 0
end = ~/diaphragm/five + {change, 0}
valvepos = [ ~/diaphragm/four, end ]
open = false
}

```

### C.3.4 Hydrolift ADM redefinitions

The following ADM script redefines the entities in the MUL animation by instantiating the entity definitions for the pump and sensor and changing the definitions of the landing and car entities.

```

%adm
entity landing(_F) {
action
  landButton{_F} == UP && sensed{_F - 1} == UP && brake == OFF -> brake = ON,
  landButton{_F} == DOWN && sensed{_F + 1} == DOWN && brake == OFF -> brake = ON,
  sensed{_F} != NIL -> landButton{_F} = OFF
}

entity car(_F) {
action
  carButton{_F} == ON -> chan1 = _F,
  chan2 == _F -> carButton{_F} = OFF
}

entity sensor(_F) {
definition
  sensed{_F} is ( pressure == _F*k && change == k ) ? UP :
                ( pressure == _F*k && change == -k ) ? DOWN : NIL
}

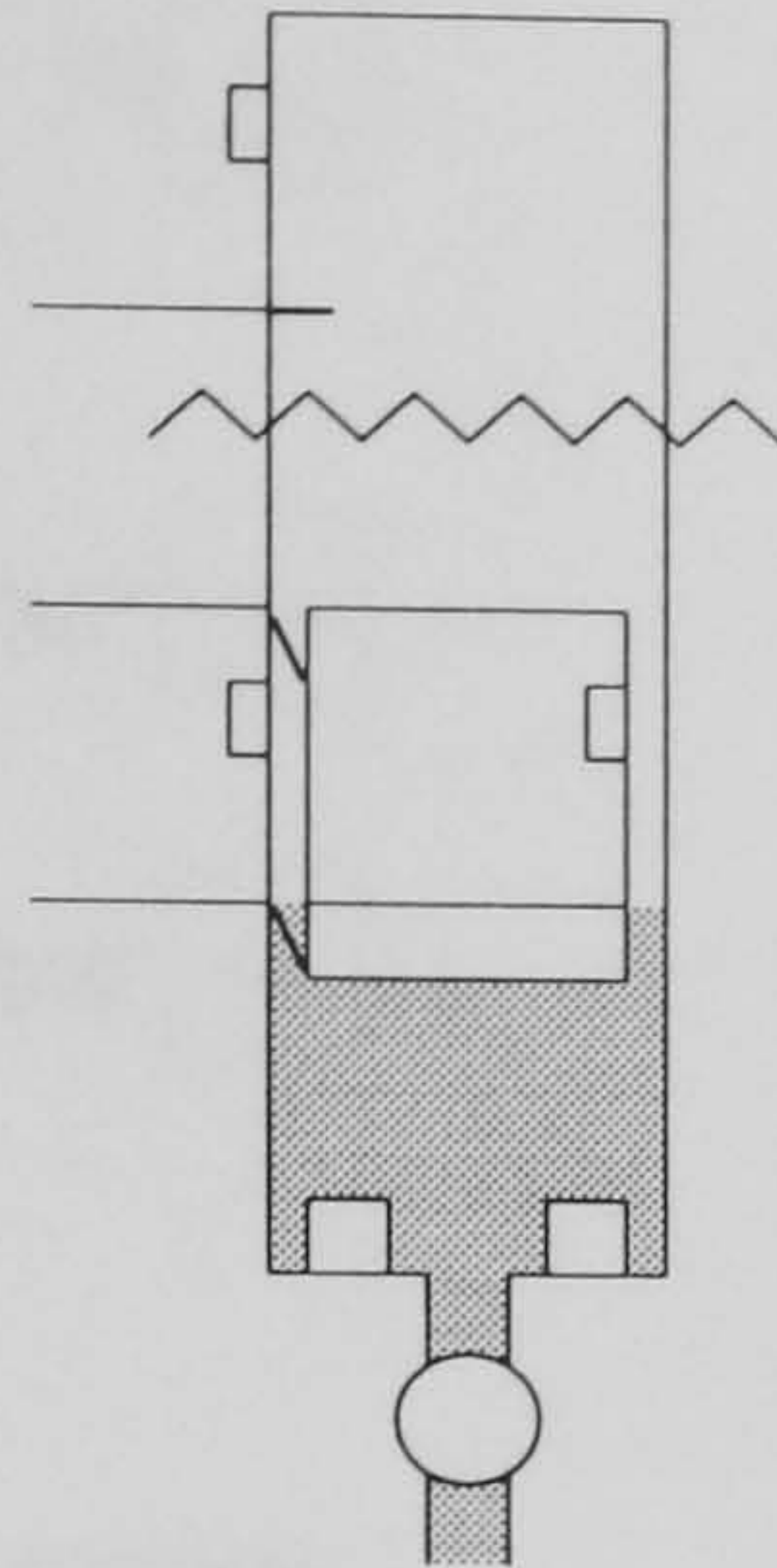
entity pump() {
definition
  k = 100,
  change is (pressure < target) ? k :
            (pressure > target) ? -k : 0
action
  target == pressure + change && brake == OFF -> brake = ON,
  change == 0 -> target = chan1*k,
  pressure == target -> chan2 = target/k,
  brake == OFF -> pressure = pressure + change,
  brake == ON && change != 0 -> brake = OFF
}

# instantiate new entities
sensed_0 = 0
sensor(1)
sensor(2)
sensor(3)
sensor(4)
sensor(5)
sensed_6 = 0
pump()

```



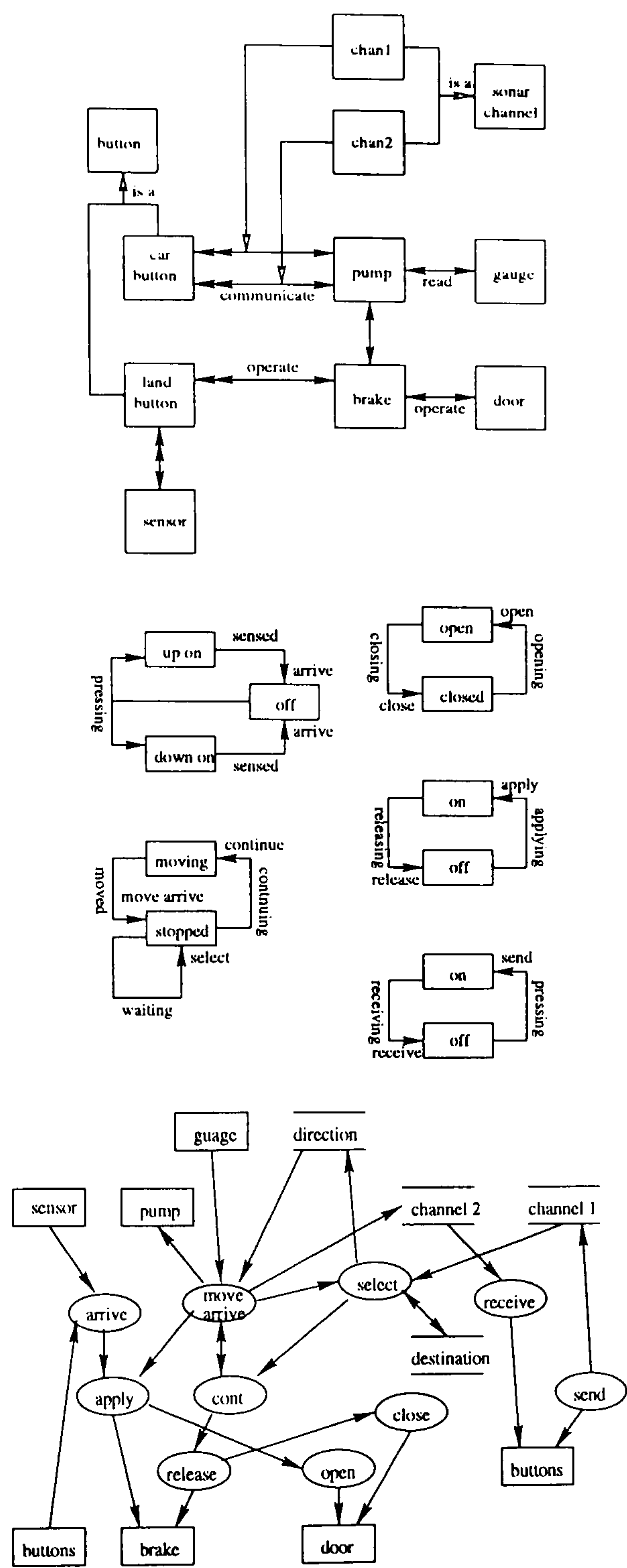
### C.3.5 Hydrolift sketch



### C.3.6 Statement of requirements and models

On each landing there is an up and a down button. In the car there is a button for each floor. Users make requests for the car to come to their landing or go to another landing by pressing these buttons. The shaft mechanism consists of sensors for detecting the direction of the car, sonar for communicating between a pump and the car, and a guage situated with the pump at the base of the shaft for detecting the pressure of the water in the shaft. The sonar has a channel from car to pump (channel 1) and a channel from pump to car (channel 2) which carry floor numbers. The pump moves the car towards a destination landing stopping whenever the brake is applied. The brake is applied whenever the car arrives at a landing requested by a user (for requests from landings the direction matters). The landing buttons are reset locally and the car buttons are reset by a signal on sonar channel 2. On arriving at the destination landing the pump selects the next destination from those transmitted on sonar channel 1. The pump releases the brake and starts the car moving again. For safety the door is opened and closed by the brake ensuring that the door is only open whilst the brake is on.





- open      open door
- close     close door
- apply     apply brake  
          generate opening
- release   generate closing  
          release brake
- receive   reset button
- send      queue destination
- select    change destination  
          and direction  
          generate waiting  
          or continuing
- arrive    reset button  
(landing) generate applying
- arrive    generate receiving  
(pump)   generate applying
- continue generate releasing  
          generate moved
- move      generate arriving  
          move car  
          generate waiting  
          or continuing



# Appendix D

## Reviews

This appendix reviews the books that form an important foundation to the work in this thesis.

### **Creative Cognition: Theory, Research, and Applications**

Ronald A. Finke, Thomas B. Ward, and Steven M. Smith.

Published by Bradford, The MIT Press, paper 1996 (hard 1992).

#### **Table of contents**

1. Introduction to Creative Cognition
2. Theoretical and Methodological Considerations
3. Creative Visualization
4. Creative Invention
5. Conceptual Synthesis
6. Structured Imagination
7. Insight, Fixation, and Incubation
8. Creative Strategies for Problem Solving

## 9. General Implications and Applications

**Cover**

*Creative Cognition combines original experiments with existing work in cognitive psychology to provide the first explicit account of the cognitive processes and structures that contribute to creative thinking and discovery. In separate chapters, the authors discuss visualization, concept formation, categorization, memory retrieval, and problem solving. They describe novel experimental methods for studying creative cognitive processes under controlled laboratory conditions, along with techniques that can be used to generate many different types of inventions.*

The review by John Richardson in the Times Higher Education Supplement praises the book for tackling a particularly difficult area of psychology:

Original and well articulated ... [A] benchmark for psychologists who are concerned to understand and explain one of the less tractable areas of human cognition. It can also be recommended as a rich source of practical ideas to anyone responsible for education and training in professions that depend on the regular exercise of creative thinking (cited in [FWS92]).

It is this practical aspect of Creative Cognition that made it a suitable basis for investigating the link between creativity and SD in this thesis. The book was used as a rich source of practical ideas by the author of this thesis whose aim was to investigate how the development of software might be construed as one of the professions “that depends on the regular exercise of creative thinking”.

The review by Stuart Sutherland of the Laboratory of Experimental Psychology, University of Sussex in NATURE begins with a general criticism of the research into creativity:

Creativity, whether in science, literature, music, painting or everyday life remains a mystery, despite the fact that psychologists are increasingly turning their attention to the topic. Creative Cognition is not



unrepresentative of their efforts. Too often they put old ideas together in imprecise ways, call the result a new theory (or model) and give it a high-sounding name - in the present case “Geneplore”, which competes with previous expressions as “Concept Specialization Model” and “Structure Mapping Theory”. The outcome is usually too commonplace to be new and too vague to be a theory [Sut93].

Sutherland clearly views Creative Cognition as neither particularly better nor particularly worse than other research into creativity, in his view, research into creativity typically results in findings that are common sense and imprecise.

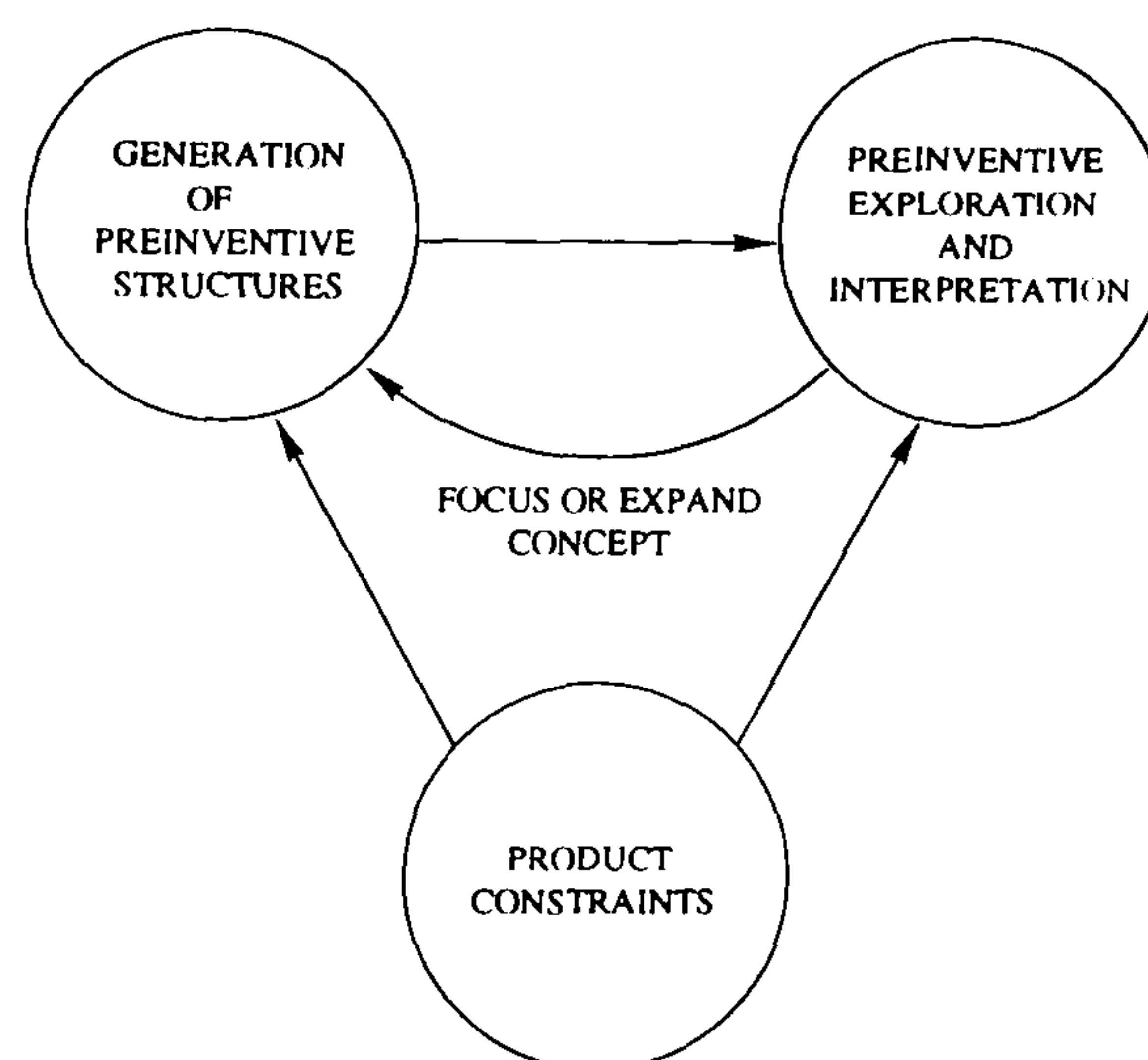


Figure D.1: Geneplore Model.

Figure D.1 shows the basic structure of the Geneplore model criticized by Sutherland. In the generative phase, one constructs mental representations called preinventive structures. These structures have various emergent properties that are exploited for creative purposes in the explanatory phase. The resulting creative cognitions can be focused or expanded according to task requirements or individual needs by modifying the preinventive structures and repeating the cycle. Constraints on the final product can be imposed at any time during the generative or exploratory phase [FWS92].

Sutherland criticizes particular aspects of the Geneplore generative phase rather than the general principle of such a phase. In particular, he does not single out the six creative properties for criticism identified in the book as being important for creative discovery or the six generative processes described in the book. It is these properties and processes that are made extensive use of in this thesis. What

Sutherland does criticize is the idea that restricted choice during the generative phase leads to increased creativity, arguing that surely this just leads to increasingly “bizarre interpretations of the restricted structures.” This particular idea of restricting choice is not pursued in connection with SD in this thesis.

As with the generative phase, Sutherland seems to have no problem with the exploratory phase and exploratory processes in principle. However, he does comment that it is not entirely clear from the book whether the exploratory phase takes place at a conscious or unconscious level. This is rather unjust since the authors state that exploration would typically occur in a “deliberate and controlled manner” and “in an organized and systematic way” which clearly suggests a conscious process. Certainly the exploratory actions described in this thesis, corresponding to the generative processes of creative cognition, are meant to be applied at a conscious level.

Sutherland returns to his theme of vagueness: “Nothing in this book is sufficiently precise to suggest a working program. The best parts of it are those concerned with well-worn findings.” Perhaps this is true, but creativity is a very difficult subject in which to be precise as recognized by Richardson. The authors clearly state their intention is to reach a balance between the “demystification of creativity” on the one hand whilst not wanting to “define creativity out of existence, or minimize it conceptually, because there really is something special about the creative mind - something that will always be surprising and innovative.” This subtle approach suited investigating the essence of EM in this thesis.

Sutherland concludes his review by saying that “one cannot help feeling that there is more to creativity than meets the authors’ eyes.” This is no doubt true because of the apparently unfathomable complexity of creativity. What the authors do manage to establish, about the structures and processes of creativity, has been used in this thesis to investigate the link between creativity and SD.

All the authors teach at Texas A & M University. Ronald A. Finke and Steven M. Smith are Associate Professors, and Thomas B. Ward is Professor of Psychology.



## Total Design: Integrated Methods for Successful Product Engineering

Stuart Pugh.

Published by Addison-Wesley, 1991.

### Table of contents

1. The Total Design Activity
2. Design Core: Market/User Needs and Demands
3. Design Core: The Product Design Specification
4. Design Core: Conceptual Design
5. Design Core: Detail Design (Technical Design)
6. Design Core: Manufacture
7. Design Core: Selling (Marketing)
8. Variations to the Total Design Activity Model
9. Design Management
10. Electronic Aids to Total Design
11. Further Methods to Assist the Design Core
12. Total Design: A Summary
13. Exercises to Illustrate the Design Core

### Cover

*Design is vital to a manufacturing company's goal of creating successful products. This book provides a framework for design whose overriding purpose is to create innovative products that satisfy the needs of the customer. Based around a core of design activities [shown in Figure D.2]*

*design is presented as a systematic and disciplined process. Features [of the book] include:*

- *a concise introduction to the total design process;*
- *a clear and simple model of design, independent of technology and discipline, allowing a structured approach to tackling design problems;*
- *numerous examples taken from a variety of fields;*
- *a chapter featuring a wide selection of design exercises.*

*The book is aimed at all students in Engineering, Industrial Design, Architecture and the professional engineer and designer, for whom it is suggested will provide a useful framework to assist their design practice.*

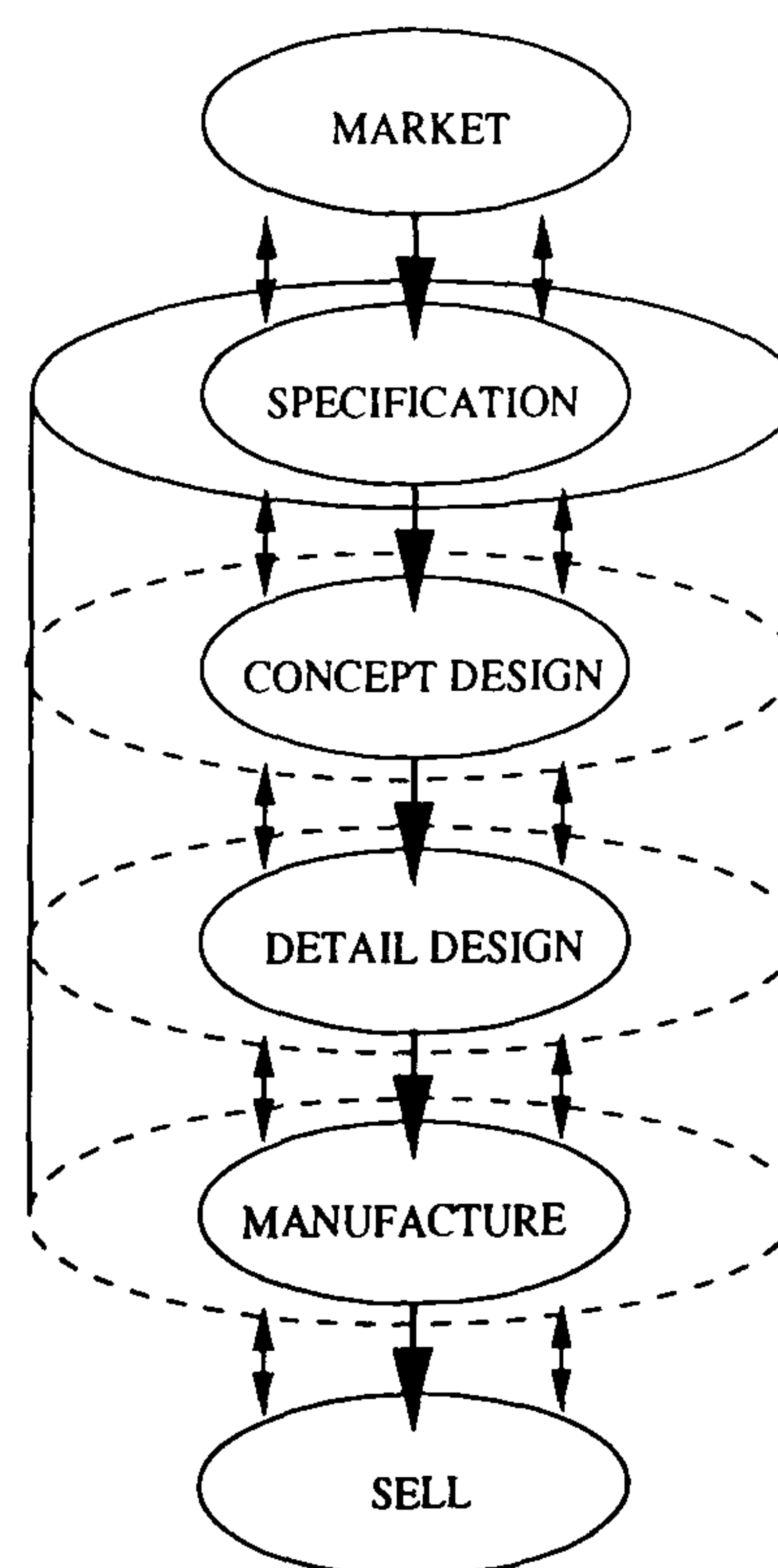


Figure D.2: Activity model for total design.

So far as I am aware, this book is unlike any other book on engineering design in that it attempts to represent the creative and analytical aspects of design and how they interrelate. Pugh states his intentions early on in the book:

This is not a book about machine element design. Neither is it a book about finite element analysis. There are already many good books in these areas. What it sets out to be is a book that defines and takes the



very complex jig-saw of the design process (like pieces of any artefact) and assembles it in a coherent and recognizable way, to give a uniform view of the picture on the box - jig-saws being more difficult to do without the guiding picture. Having said this, detailed analytical and technical topics are essential to the successful design of any product, and these will be considered and fitted in as the picture unfolds ([Pug91] p.vii).

This balanced view of design was used in this thesis as a model for investigating the roles of creativity and analysis in EM and how creativity might be introduced into SD.

When he wrote the book Stuart Pugh was head of the Design Division at the University of Strathclyde where he taught design to all undergraduate engineers. His career in industry included service as Chief Engineer and then Divisional Manager with the English Electric Company, as well as numerous design positions with the British Aircraft Corporation and the Marconi Company. He consulted with numerous companies in the United States, including DIGITAL and General Motors, on some of their most successful products.

## **Creating Innovative Products Using Total Design: The Living Legacy of Stuart Pugh**

Stuart Pugh, edited by Don Clausing and Ron Andrade.  
Published by Addison-Wesley, 1996.

### **Table of contents**

1. Design in Education and Industry
2. Design Process and Philosophy
3. Design Techniques and Methods
4. CAD and Knowledge-Based Engineering
5. Design Teams, Management, and Creative Work

- 6. Design for X
- 7. Design Research
- 8. Total Design: Summary of the whole

This book is unusual in that it was compiled by the editors from Pugh's collected works after his untimely death in October 1993. The book is essentially an organized collection of papers written by Pugh and accounts of conference presentations given by Pugh before and after the publication of his book entitled "Total Design: Integrated Methods for Successful Product Engineering" in 1990. The book provides an insight into the ideas of Pugh from the perspectives of Clausing and Andrade and how the ideas of Pugh have been adopted by the design community in general.

Clausing writes about Pugh as being neither wholly an academic nor an industrialist in the preface to his book:

Stuart Pugh was one of the great leaders of product development (total design) methodology and practice ... Very few people have duplicated Stuart's experience of spending almost half of his career in successful industrial practice and then the remainder of his career in a university. Through this dual career Stuart developed a comprehension of and insights into total design that went far beyond those supported by the more traditional monolithic career, whether in industry or academia. These profound insights culminated in Stuart's book "Total Design" published in 1990 (Clausing [Pug96] p.xix).

Pugh's concern was that the academic teaching of design was aloof from industrial practice, while industrial practice suffered from the lack of reflective structuring that can be achieved in the university: "The symbiosis between design education in universities and design practice in industry is the foundation of Stuart Pugh's path to design success ... total design is the great integrator of the engineering curriculum ... total design is the integrator between the academy and industry" (Clausing [Pug96] p.1). A concern that is being increasingly echoed within the software industry [Lew95].



Pugh saw one of the reasons for the gap between academia and industry being the tendency of researchers to disintegrate and simplify the inherently integrated and complex design process. Pugh points out that in considering research we must make the following three distinctions:

- total design and partial design;
- static products and dynamic products;
- technology-specific methods and generic methods.

Much research into design processes and methods is primarily applicable to the partial design of static products in some specific technology set. Such methods can be useful in their particular domain. However, they are best viewed as subsets of total design, providing the right details in the context of the more important decisions that have been made by applying the generic methods that Pugh emphasized.

The formulation of the design activity model was born out of the need to give an definition of design that captured its complexity. This definition of design has been adopted by SEED (Sharing Experience in Engineering Design - a multi-disciplinary organization comprising lecturers in engineering design throughout the UK) “quite simply because design practitioners relate to it” (Pugh [Pug96] p.xxxii):

A perennial problem that arises at design conferences and discussions is understanding just what is meant by design and design engineering ... I described design as a highly manipulative activity in which the designer has to continuously and simultaneously pay attention to and balance several factors that impinge and influence design ... a step further was the proposition of the design activity model [shown in Figure D.2] ... We made significant progress, and this was recognised by Sharing Experiences in Engineering Design (SEED), an organisation based in the U.K. academia at varying levels. This model now forms the basis of design teaching in more than eighty U.K. institutions (Pugh [Pug96] p.xxviii).

Part of the success of the model is that it provides a guide to rather than prescribes how design should be done: “I regard the model’s structure as being analogous to

a child's climbing frame: it provides the framework on which to climb, it imparts confidence and safety, yet it doesn't prescribe or predetermine the methods by which the child gets to the top of the frame or indeed around inside it" (Pugh [Pug96] p.50). This is consonant with the view of EM as a framework for systems modelling in this thesis.

Dr. Clausing is the Xerox Fellow in Competitive Product Development at M.I.T., and Dr. Andrade is Professor of Product Development and Quality Management at the Federal University of Rio de Janeiro, Brazil.